DIPLOMARBEIT

Thema:

Selektion und Evaluierung datenbankbasierter Persistenzlösungen für Java-Objekte basierend auf JDO (Java Data Objects)

Bearbeiter:

Timo Koster

Betreuung:

Prof. Dr. Andreas Künkler, FH Trier

Dipl.-Inform. Andreas Drebinger, Siemens AG Erlangen

Angewandte Informatik

Fachhochschule Trier

Fachbereich Design und Informatik

Studiengang Angewandte Informatik

Kurzfassung

JDO (Java Data Objects) wurde im März 2002 von SUN verabschiedet und stellt eine Spezifikation für transparente Persistenz von Objekten unter Java dar.

Unabhängig vom verwendeten Speichermedium, z.B. relationale oder objektorientierte Datenbank oder die Speicherung in Dateien, stellt sie dem Anwender eine genormte API (Application Programming Interface) zur Verfügung.

In dieser Diplomarbeit werden verschiedene JDO-Implementierungen und objektrelationale Abbildungswerkzeuge anhand eines konkreten Anwendungsfall untersucht und gegenübergestellt.

Die Diplomarbeit ist im wesentlichen in drei Teile gegliedert. Der erste Teil ist eine Untersuchung und Beschreibung der JDO-Spezifikation. Im zweiten Teil werden der Anwendungsfall und die Implementierung der erstellten Prototypen erläutert. Der dritte Teil stellt die Ergebnisse der Evaluierung dar.

Abstract

JDO, launched by SUN in March 2002, represents a specification of transparent persistence of objects in Java.

JDO provides a standardized API for making objects persistent, independent of the underlying storage system like relational or object-oriented databases.

In this paper different JDO-implementations and object-relational mapping tools will be discussed and compared. The performance of different operations will be evaluated based on a prototype application.

This paper consists of three parts, the first one forms a theoretic survey and a description of the JDO specification. In the second part, the application and the implementation of the prototype will be discussed. Finally, the third part of this paper describes the results of the evaluation.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Erlangen, Juni 2003

Timo Koster

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die zum Gelingen dieser Diplomarbeit beigetragen haben.

Ich danke Herrn Dipl.-Inform. Andreas Drebinger für die Vermittlung der interessanten Themenstellung sowie seiner Betreuung während dem gesamten Projektverlauf.

Herrn Prof. Dr. Andreas Künkler für die Betreuung und Unterstützung dieser Diplomarbeit.

Allen Mitarbeitern des Leittechnik-Projektes der Siemens AG Erlangen, besonders Gerhard Arzberger und Rainer Stumpf, die stets ein offenes Ohr für Fragen hatten.

Ein besonderer Dank gilt meinen Eltern, die mir finanziell und die ganzen Jahre mit Rat und Tat zur Seite standen.

Chefarzt Dr. Joachim Vogt für die Vermittlung der Tätigkeit als studentische Aushilfe im Schlaflabor des Brüderkrankenhauses Trier. Die dadurch entstandenen nächtlichen „Lernattacken“ ermöglichten mir ein zügiges Beenden des Studiums.

Solveig Schmid für das Korrekturlesen der englischen Kurzfassung und Julia Bunke für das Korrekturlesen der gesamten Diplomarbeit.

Inhalt

1. Einleitung....14
1.1 Konventionen....15
2. Analyse....15
2.1 Überblick....15
2.1.1 Dateisystem und Serialisierung....15
2.1.2 Relationale Datenbank Management Systeme (RDBMS)....16
2.1.3 Enterprise JavaBeans (EJB)....17
2.1.4 Objektrelationale Datenbank Management Systeme (ORDBMS)....18
2.1.5 Objekt-relationale Abbildungen bzw. Mapping-Tools....18
2.1.6 Objektorientierte Datenbank Management Systeme (OODBMS)....19
2.2 Java Data Objects....19
2.2.1 JDO-Architektur....21
2.2.1.1 Umgebungen von JDO....21
2.2.1.2 Erweiterungsvorgang (Enhancement)....22
2.2.1.3 Grundlegende Begriffe....23
2.2.1.4 Das JDO-Identitätenkonzept....25
2.2.2 Klassen, Interfaces und Packages....27
2.2.3 Persistenz-Deskriptor....28
2.2.4 Objekt-Modell....28
2.2.4.1 Transparente Persistenz....28
2.2.4.2 Unterstützte Datentypen und Modifikatoren von JDO....29
2.2.4.3 Vererbung unter JDO....31
2.2.4.4 Zustände der JDO-Instanzen....32
2.2.4.5 Instanzen-Callbacks....34
2.2.5 Transaktionen unter JDO....35
2.2.5.1 pessimistische Transaktionen (JDO-Standard)....35

2.2.5.2 optimistische Transaktionen (optionale Eigenschaft).....35
2.2.5.3 Wichtige Methoden der Schnittstelle Transaction.....36
2.2.6 Abfragen mittels JDOQL..... 37
2.2.7 Ausnahmebehandlung unter JDO.....38
2.3 Selektionskriterien der JDO-Implementierungen.....40
2.4 JDO-Implementierungen (Frameworks).....41
2.4.1 Signsoft intelliBO V3.1.....41
2.4.2 Hibernate V1.2.3.....41
2.4.3 CASTOR JDO..... 42
2.4.4 Solarmetric Kodo JDO V2.4.3..... 42
2.4.5 Libelis Lido V1.4..... 42
2.5 Kriterien für die Evaluierung.....43
2.5.1 Integrationsaufwand für bereits existierende Klassen.....43
2.5.2 Performance und Ressourcenverbrauch..... 43
2.5.3 Laufzeitschwankungen.....43
2.5.4 Integration objektorientierter Paradigmen.....43
2.5.4.1 Vererbung.....43
2.5.4.2 Assoziationen..... 43
2.5.4.3 Komposition.....44
2.5.5 Graphische Entwicklungsumgebung.....44
2.5.6 Abfragesprache(n).....44
2.5.7 Unterstützte Systemumgebungen.....44
3. Prototyp.....44
3.1 Systemumgebung.....44
3.1.1 Die Datenbank MySQL.....45
3.1.1.1 Warum MySQL?.....45
3.1.1.2 Der Tabellentyp innoDB.....45
3.1.2 Die Datenbank Hypersonic SQL (HSQL)..... 45

3.2 Klassendiagramm und Implementierung....46
3.2.1 Implementierung intelliBO....48
3.2.1.1 Entity-Relationship-Modell....49
3.2.1.2 Persistenz-Deskriptor am Beispiel Primary.jdo....50
3.2.1.3 Installation + Benutzeranleitung....53
3.2.2 Implementierung Hibernate....53
3.2.2.1 Wichtige Klassen und Schnittstellen der Hibernate-API....54
3.2.2.2 Entity-Relationship-Modell....55
3.2.2.3 Persistenz-Deskriptor....57
3.2.2.4 Installation + Benutzeranleitung....58
3.2.3 Implementierung Castor....59
3.2.3.1 Wichtige Klassen und Schnittstellen der Castor-API....59
3.2.3.2 Entity-Relationship-Modell....61
3.2.3.3 Persistenz-Deskriptor....63
3.2.3.4 Installation + Benutzeranleitung....64
3.2.4 Implementierung Solarmetric Kodo JDO V2.3.4....65
3.2.4.1 Entity-Relationship-Modell....67
3.2.4.2 Persistenz-Deskriptor....68
3.2.4.3 Installation + Benutzeranleitung....70
3.2.5 Implementierung Libelis Lido V1.4....71
3.2.5.1 Entity-Relationship-Modell....72
3.2.5.2 Persistenz-Deskriptor....74
3.2.5.3 Installation + Benutzeranleitung....75
4. Ergebnisse und Fazit....75
4.1 Signsoft intelliBO....75
4.1.1 Automatisches Erstellen der zugehörigen Tabellen....76
4.1.2 Integrationsaufwand für bereits existierende Klassen....77

4.1.3 Performance und Ressourcenverbrauch..... 77
4.1.3.1 Insert-Operation.....79
4.1.3.2 Delete-Operation.....79
4.1.3.3 Query-Operation.....80
4.1.3.4 Update-Operation.....81
4.1.3.5 Speicherverbrauch.....82
4.1.3.6 Performance und Ressourcenverbrauch intelliBO V3.2.....82
4.1.4 Vererbung.....83
4.1.5 Assoziationen..... 84
4.1.6 Komposition.....85
4.1.7 Graphische Entwicklungsumgebung.....85
4.1.8 Abfragesprache(n).....86
4.1.9 Unterstützte Systemumgebung.....86
4.2 Hibernate.....86
4.2.1 Automatische Generierung des Persistenz-Deskriptors.....86
4.2.2 Automatisches Erstellen der zugehörigen Tabellen.....87
4.2.3 Integrationsaufwand für bereits existierende Klassen.....87
4.2.4 Performance und Ressourcenverbrauch..... 87
4.2.4.1 Insert-Operation.....88
4.2.4.2 Delete-Operation.....89
4.2.4.3 Query-Operation.....90
4.2.4.4 Update-Operation.....91
4.2.4.5 Speicherverbrauch.....92
4.2.4.6 Performance unter Verwendung der Datenbank HSQL.....92
4.2.5 Vererbung.....93
4.2.6 Assoziationen..... 93
4.2.7 Komposition.....93
4.2.8 Graphische Entwicklungsumgebung.....93

4.2.9 Abfragesprache(n)....94
4.2.10 Unterstützte Systemumgebung....94
4.3 Castor JDO....94
4.3.1 Integrationsaufwand für bereits existierende Klassen....94
4.3.2 Hilfswerkzeuge....94
4.3.3 Performance und Ressourcenverbrauch....95
4.3.3.1 Insert-Operation....95
4.3.3.2 Delete-Operation....96
4.3.3.3 Query-Operation....97
4.3.3.4 Update-Operation....98
4.3.3.5 Speicherverbrauch....99
4.3.3.6 Performance unter Verwendung der Datenbank HSQL....99
4.3.3.7 Vererbung....100
4.3.3.8 Assoziationen....100
4.3.3.9 Komposition....100
4.3.3.10 Abfragesprache....100
4.3.3.11 Unterstützte Systemumgebung....101
4.4 Solarmetric Kodo JDO V2.3.4....101
4.4.1 Automatisches Erstellen der zugehörigen Tabellen....101
4.4.2 Automatische Generierung des Persistenz-Deskriptors....101
4.4.3 Weitere Hilfsprogramme....101
4.4.4 Integrationsaufwand für bereits existierende Klassen....102
4.4.5 Performance und Ressourcenverbrauch....102
4.4.5.1 Insert-Operation....102
4.4.5.2 Delete-Operation....103
4.4.5.3 Query-Operation....104
4.4.5.4 Update-Operation....105

4.4.5.5 Speicherverbrauch....106
4.4.5.6 Performance unter Verwendung der Datenbank HSQL....106
4.4.6 Vererbung....107
4.4.7 Assoziationen....108
4.4.8 Komposition....108
4.4.9 Graphische Entwicklungsumgebung....108
4.4.10 Abfragesprache(n)....108
4.4.11 Unterstützte Systemumgebung....108
4.5 Libelis Lido V1.4....109
4.5.1 Automatisches Erstellen der zugehörigen Tabellen....109
4.5.2 Integrationsaufwand für bereits existierende Klassen....109
4.5.3 Performance und Ressourcenverbrauch....109
4.5.3.1 Insert-Operation....109
4.5.3.2 Delete-Operation....110
4.5.3.3 Query-Operation....111
4.5.3.4 Update-Operation....112
4.5.3.5 Speicherverbrauch....113
4.5.3.6 Performance unter Verwendung der Datenbank HSQL....114
4.5.4 Vererbung....115
4.5.5 Assoziationen....115
4.5.6 Komposition....115
5. Zusammenfassung....115
6. Anhang....120
6.1 Abbildungsverzeichnis....120
6.2 JDO-Implementierungen-Quellen....121
6.3 Ergebnisse der Internetrecherche....122
6.4 Installation und Konfiguration von MySQL....123

6.5 Installation und Konfiguration von Hypersonic SQL (HSQL).....124

6.6 Persistenz-Deskriptor Hibernate.....125

6.7 SQL-Skript für die Datenbank hibernate.....127

6.8 Persistenz-Deskriptor Castor JDO.....128

6.9 SQL-Skript für die Datenbank castordiplom.....131

6.10 Persistenz-Deskriptor Kodo JDO.....132

6.11 SQL-Skript für die Datenbank kodoDiplom.....134

6.12 Persistenz-Deskriptor Lido.....135

6.13 SQL-Skript für die Datenbank lidodiplom.....137

6.14 DTD-File.....138

6.15 Quellenverzeichnis 139

1. Einleitung

Die dauerhafte Speicherung von Daten ist ein wichtiger Bestandteil vieler Softwareprojekte. Speziell für Java-Objekte existieren verschiedene Ansätze mit spezifischen Vor- und Nachteilen. Ziel dieser Diplomarbeit ist es, anhand geeigneter Prototypen standardisierte Persistenzlösungen für Java-Objekte in Datenbanken zu untersuchen. Die Anforderungen für die zu erstellenden Prototypen resultieren aus einem laufenden Projekt im Leittechnikumfeld des Geschäftsbereich Power Generation der Siemens AG Erlangen. Hierbei sollen insbesondere JDO-basierte (Java Data Objects) und objekt-relationale Abbildungs-Technologien evaluiert und Vor- bzw. Nachteile aufgezeigt werden. Die JDO-Spezifikation stellt eine einheitliche API (Application Programming Interface) für persistentes Speichern von Java-Objekten zur Verfügung. Persistenzframeworks stellen eine transparente Persistenz für den Anwendungsentwickler dar, d.h. sie kümmern sich automatisch um die Transformation der Java-Objekte auf relationale Datenbanken. Hinsichtlich Integrationsaufwand für bereits existierende Klassen, Performance, Ressourcen-Verbrauch, Laufzeitschwankungen, Integration objekt-orientierter Paradigmen, usw. wurden folgende Frameworks näher untersucht:

- JDO-Implementierung intelliBO (Fa. Signsoft, [Sign02])
- JDO-Implementierung Kodo (Fa. Solarmetric, [Kodo03])
- JDO-Implementierung Lido (Fa. Libelis, [Lido03])
- Objekt-Relational-Mapper Castor JDO (open-source, [Cast03])
- Objekt-Relational-Mapper Hibernate (open-source, [Hibe03])

In Kapitel 2 werden Möglichkeiten zur Persistenz in Java vorgestellt, sowie eine Einführung in Java Data Objects (JDO) gegeben. Die Abschnitte 2.3 und 2.5 beschreiben die Selektions- und Evaluierungskriterien. Der Abschnitt 2.4 stellt die untersuchten JDO-Implementierungen bzw. Objekt-Relationale-Abbildungswerkzeuge dar. Das Kapitel 3 beschreibt den zu untersuchenden Anwendungsfall und den erstellten Prototyp anhand eines Klassendiagramms. Das Kapitel ist so aufgebaut, dass dort die Änderungen des zu Beginn erstellten Prototyps für den jeweiligen Framework, dass zugehörendene Entity-Relationship-Modell und spezifische Eigenschaften der Persistenz-Deskriptoren erläutert werden. Kapitel 4 beschreibt unabhängig von den anderen Produkten die erzielten Ergebnisse, so dass der Leser die Möglichkeit hat, nur Ergebnisse über die ihn interessierende JDO-Implementierung oder des Objekt-Relationalen-Abbildungswerkzeuges zu lesen. In Kapitel 5 werden oben genannte Frameworks gegenübergestellt und ein Fazit getroffen. Im Anhang sind die erstellten Persistenz-Deskriptoren, die erstellten Datenbank-Schemata und Installationsanleitungen zu finden.

1.1 Konventionen

Folgende Konventionen werden in dieser Diplomarbeit verwendet:

`Schreibmaschinentext`
für Programmcode, Klassen- und Schnittstellennamen.

2. Analyse

In diesem Kapitel werden gängige Persistenzlösungen für Java-Objekte vorgestellt und Gründe genannt wieso ein neuer Standard names JDO gefordert wurde. Des weiteren wird ein kurzer Überblick über die JDO-Spezifikation gegeben, die Selektions- bzw. Evaluierungskriterien sowie die zu untersuchenden JDO-Implementierungen bzw. Objekt-Relationale-Abbildungswerkzeuge näher erläutert.

2.1 Überblick

Dieser Abschnitt beschreibt die zur Verfügung stehenden Möglichkeiten zur Speicherung von Objekten in Java. Es wird kurz die jeweilige Technik erläutert sowie deren spezifischen Vor- und Nachteile aufgelistet. Generell versteht man unter Persistenz, dass Objekte nicht nur zur Laufzeit, sondern auch nach dem Beenden der Java Virtual Machine (JVM) zur Verfügung stehen.

2.1.1 Dateisystem und Serialisierung

Eine sehr einfache Möglichkeit zur Speicherung von Objekten unter Java besteht darin, dass man die Attribute der Objekte direkt in Dateien speichert, z.B. in `Properties`. Dies ist aber nur zur Speicherung einfacher Informationen geeignet, wie z.B. URL einer Datenbank, Username, Passwort usw..

Unter Serialisierung versteht man, dass das zu speichernde Objekt und alle referenzierten Objekte in einen Bytestrom geschrieben werden. Dieses kann dann lokal auf der Platte in einer Datei abgelegt oder z.B. mittels RMI[1] übertragen werden, und bei Bedarf wird aus dieser Datei die Objektstruktur wiederhergestellt. Ein Objekt kann serialisiert werden, wenn die Klasse die leere Schnittstelle `java.io.Serializable` implementiert.

Vorteile:

- sehr einfache Handhabung: der Anwender braucht sich um die Codierung bzw. Decodierung des Java-Objektmodells keine Gedanken zu machen. Diese Technik verläuft vollkommen transparent im Hintergrund
- fester Bestandteil ab der Java 1.1 Version

[1] Remote Method Invocation: ist ein rechnerübergreifendes Objekt-zu-Objekt-Kommunikationskonzept in Java. Es erlaubt auf einem anderen Rechner bzw. in einer anderen JVM Methoden von Objekten aufzurufen, als wären diese auf dem eigenen Rechner.

Nachteile:

- es steht kein Transaktionskonzept zur Verfügung
- Mehrbenutzerzugriff wird nicht unterstützt
- Anfragen zum Finden einzelner Objekte werden nicht unterstützt: es muss immer die komplette Objektstruktur aus der Datei eingelesen werden da keine normierte Abfragesprache, wie z.B. SQL[1] bei relationalen Datenbanksystemen existiert.
- ändert sich ein einzelnes Attribut eines Objektes, so muss immer das ganze Objekt serialisiert bzw. deserialisiert werden

2.1.2Relationale Datenbank Management Systeme (RDBMS)

1970 wurde das relationale Modell, basierend auf den mathematischen Konzepten der Relationen-Algebra, von Edgar F. Codd definiert. Es ist das weit verbreiteste Datenbankmodell. Der Grund liegt in seiner Einfachheit, d.h. es gibt nur eine Datenstruktur. Diese besteht aus einer Tabelle mit Zeilen und Spalten, die Daten bestimmter Typen enthalten. Die zu speichernden Objekte werden in ihre Attribute zerlegt und durch SQL-Anweisungen in einer relationalen Datenbank abgespeichert. Die ursprünglichen Objekte erhält man, indem man ein neues Objekt erzeugt und diesem die aus der Datenbank ausgelesenen Attributwerte zuweist.

Vorteile:

- Transaktionskonzept wird von den meisten relationalen Datenbanksystemen unterstützt
- Bei normalisierten[2] Modellen ist Datenkonsistenz garantiert, da Änderungen nur an einer Stelle vorgenommen werden
- effiziente Verwaltung großen Mengen einfach strukturierter Daten
- es existieren APIs wie z.B. JDBC[2] und SQLJ[3], die einen standardisierten Zugriff auf relationale Datenbanksysteme von unterschiedlichen Herstellern erlauben

Nachteile:

- Die Persistenzlogik (JDBC) ist fest im Programmcode verankert (keine Transparenz)
- Der Anwender muss die Abfragesprache SQL erlernen

[1] Structured Query Language: ist eine nicht-prozedurale, mengenorientierte und plattformunabhängige Sprache, durch die beschrieben wird, wie Daten abzurufen, zu löschen oder einzufügen sind

[2] Normalisierung: basierend auf Normalformen dient der Vorgang der Normalisierung zur Verfeinerung eines relationalen Schemas um Mehrdeutigkeiten und Redundanzen zu entfernen

[2] Java Database Connectivity: eine herstellerneutrale Datenbankschnittstelle, über die eine Verbindung zur Datenbank bzw. Abfragen mittels SQL an die Datenbank möglich sind, ohne sich um die spezifischen Eigenschaften des verwendeten Datenbanksystem kümmern zu müssen

[3] Structured Query Language for Java: dies ist eine erweiterte SQL-Abfragesprache speziell für JAVA. Dabei werden durch einen SQL-Precompiler die SQL-Anweisungen direkt im Programmcode durch Java-Code ersetzt

- Objektorientierte Paradigmen werden nicht unterstützt, so dass der Programmierer eine Transformation vom objektorientiertem Modell in das relationale Modell von Hand durchführen muss. Hierbei entstehen häufig Fehler und ein höherer Implementierungsaufwand.

- geringe Anzahl von Basisdatentypen

- da durch den Normalisierungsprozess das Objekt aufgesplittet wird, und somit durch Verlinkung über Fremdschlüssel auf mehrere Tabellen verteilt wird, müssen bei Zugriff auf das gesamte Objekt aufwendige Verbundoperationen (Joins) durchgeführt werden, was die Performanz der Datenbank negativ beeinflusst

- der Wechsel zu einem anderen relationalen Datenbanksystem kann zu Problemen führen, da die Hersteller teilweise unterschiedliche Datentypen verwenden

2.1.3Enterprise JavaBeans (EJB)

In der Enterprise JavaBeans-Spezifikation, die Bestandteil der J2EE[1]-Gesamtspezifikation ist, sind weitere Persistenzlösungen eingebettet. Durch Komponentenorientierung wird ein hoher Grad an Widerverwendbarkeit erreicht und der Einsatz von Applikationsservern[2] bietet dem Entwickler transparente Mechanismen für z.B. Datenbankzugriff, Transaktionskonzept usw.

Dieser Abschnitt beschreibt nicht die EJB-Spezifikation, sondern richtet sein Augenmerk darauf, wie der Einsatz von Komponenten für die Persistenz in Java in Frage kommt. Ein Mechanismus dafür nennt sich container-managed-persistence (CMP): Durch Implementierung von Schnittstellen wird aus einer Klasse eine Komponente (Entity-Bean), und eine XML-Beschreibungsdatei (Deskriptor) definiert die Attributwerte der Klasse, die persistent abgespeichert werden sollen. Dadurch erhält man eine Komponente, die in einen Applikationsserver eingefügt werden kann, wodurch dann automatisch unter anderem die Funktionalität für die Persistenz bereitgestellt wird.

Vorteile:

- J2EE ist ein Standard, d.h. die Interoperabilität zwischen verschiedenen Herstellern der Applikationsserver ist garantiert

- Entwickler muss sich nicht um Datenbankzugriffe und Transaktionen kümmern, dieser werden automatisch vom Container[3] zur Verfügung gestellt

[1] Java 2 Enterprise Edition: ist eine Erweiterung der Java 2 Standard Edition zur Entwicklung und zum Betrieb unternehmensspezifischer Applikationen

[2] Applikationsserver: dient zur Verwaltung und Betrieb von Applikationen bzw. Teilapplikationen und stellt anderen Teilnehmern im Netzwerk die auf ihm gespeicherten Applikationen zur Verfügung.

[3] Befindet sich innerhalb eines Applikationservers. Der Container ist für die Verwaltung der Entity-Beans zuständig und bietet zuvor genannte Funktionalität an

- Komponenten sind wieder verwendbar

Nachteile:

- Das Erlernen eines sehr komplexen Komponentenmodells ist nötig
- Nicht transparent: Persistenzlogik teilweise in den Komponenten enthalten

2.1.4 Objektrelationale Datenbank Management Systeme (ORDBMS)

Durch Unterstützung von objektorientierten Konzepten sind sie eine Weiterentwicklung von relationalen Datenbanksystemen. Die Idee ist, komplexe Objekte in Relationen zu speichern und Anfragen direkt auf Objekte zu ermöglichen, so dass nicht erst das komplette Objekt ausgelesen werden muss. Durch den SQL99-Standard werden objekt-relationale Erweiterungen eingeführt, ohne bewährte SQL-Konzepte aufzugeben.

Vorteile:

- Die Abbildung Objekt $\leftrightarrow$ Datenbank entfällt
- Erweiterung der Basistypen → Möglichkeit zur Definition eigener Datentypen
- Transaktionskonzept wird unterstützt

Nachteile:

- geringe Transparenz, da die Persistenzlogik fest im Programmcode verankert ist
- Kenntnisse von SQL werden benötigt

-

2.1.5 Objekt-relationale Abbildungen bzw. Mapping-Tools

Tools übernehmen die Abbildung von der Objektstruktur auf die Tabellen, und speichern diese Objekte in relationalen Datenbanken. Dazu können sie automatisch ein passendes SQL-Schema[1] generieren, oder aus einem benutzerdefinierten Mapping ein SQL-Schema erzeugen.

Vorteile:

- Entwickler braucht sich nur um das Mapping zu kümmern

Nachteile:

[1] SQL-Schema: legt die Struktur der zu speichernden Datenobjekte, die Namen der Tabellen und Attribute mit zugehörenden Datentypen fest

- keine gemeinsame API zwischen Herstellern, d.h. die Programme bleiben streng an einen Hersteller gebunden, was bei einem Herstellerwechsel automatisch zu hohen Entwicklungs- bzw. Wartungskosten führt

2.1.6 Objektorientierte Datenbank Management Systeme (OODBMS)

1991 wurde die Object Database Management Group (ODMG) gegründet. Es ist eine Gruppierung verschiedener Hersteller von objektorientierten Datenbanken, die sich zum Ziel gesetzt haben, einen Standard für die objektorientierte Datenhaltung zu definieren. Der aktuelle ODMG-Standard (Version 3.0, Jahr 2000) beschreibt ein sprachneutrales Objektmodell, eine Objektanfragesprache (OQL)[1] und unabhängige Sprachanbindungen für C++, Java und Smalltalk. Ziel ist es, die Objekte mit ihren Beziehungen untereinander und ihren Methoden 1:1 in der Datenbank zu speichern.

Vorteile:

- Objekte müssen nicht wie bei RDBMS in ihre Attribute zerlegt werden, da das Objekt als Ganzes gespeichert wird

- Objektorientierte Paradigmen werden von OODBMS unterstützt

- Transaktionskonzept wird unterstützt

Nachteile:

- Erlernen einer speziellen Anfragesprache (OQL)

- Zugriff auf persistente Objekte bzw. deren Attribute kann ausschließlich innerhalb von Transaktionen geschehen

- Wechsel des OODBMS kann zu Problemen führen, da ODMG kein Standard für das Transaktionskonzept definiert, so dass jeder Hersteller seine eigene API zur Verfügung stellt

- Da mehr Informationen in der Datenbank hinterlegt werden müssen, haben OODBMS einen höheren Speicherverbrauch als RDBMS

2.2 Java Data Objects

Die in Abschnitt 2.1 erwähnten Nachteile der Persistenzmöglichkeiten von Java-Objekten gaben Anstoß zur Entwicklung eines neuen Standards für transparente Persistenz in Java. Es soll eine Java-API zur plattform- und datenspeicherunabhängigen Persistenz von Objekten entstehen. 1999 schlossen sich führende Java-Experten und Datenbank-Hersteller, darunter SUN, IBM und Oracle, unter Führung von Craig Russell der Firma SUN zusammen, um diesen neuen Standard zu spezifizieren. Im Juli 1999 entstand der Java Specification Request for JDO (JSR12). Im Mai 2000 wurde ein Vorabentwurf und ein Jahr später die Version 1.0 von JDO veröffentlicht. Im März 2002 wurde diese Version als offizieller Standard

[1] Object Query Language: ermöglicht Zugriff auf objektorientierte Datenbanken. Sie kann als eigenständige, interaktive Datenbanksprache oder aber eingebettet in verschiedenen Programmiersprachen benutzt werden.

deklariert. Zur Zeit wird an der Version 1.0.1 von JDO gearbeitet, um bekannt gewordene Fehler zu beseitigen und Verbesserungen hinzuzufügen.

Das JDO-Package javax.jdo ist frei von SUN Microsystems Inc. unter http://access1.sun.com/jdo/ zu erhalten. Es beinhaltet größtenteils Schnittstellen-Definitionen sowie einige wenige Implementierungen konkreter Klassen (JDOHelper, JDO-Exceptions, usw.). Durch die Definition der Schnittstellen wird ein standard-konformer Weg zur transparenten Persistenzlösung unter Java geschaffen.

Das JDO-Package ist allein nicht lauffähig. Die jeweiligen JDO-Anbieter müssen mindestens die durch die Spezifikation beschriebenen Basiseigenschaften[1] bzw. Schnittstellen von JDO implementieren, und bieten dadurch eine sogenannte standard-konforme JDO-Implementierung (Persistenz-Framework) als open-source oder als kommerzielle Variante an. Des weiteren definiert die Spezifikation noch optionale Eigenschaften von JDO, welche im Ermessen des jeweiligen JDO-Anbieters liegen, ob er diese implementiert. In Anhang 6.2 ist eine Auflistung verschiedener open-source bzw. kommerzieller JDO-Implementierungen zu finden. Diese wurden während der Analyse anhand einer Internet-Recherche näher untersucht, ob sie zur Auswahl der für diese Diplomarbeit zu evaluierenden[2] JDO-Implementierungen gehören. Dort werden auch die Selektionskriterien erläutert.

Im folgenden werden die Ziele der Spezifikation bzw. die Vorteile von JDO aufgelistet:

- Definition von Schnittstellen zur persistenten Speicherung von Java-Objekten bzw. den transparenten[3] Zugriff auf alle Arten von Datenspeichern, so dass der Anwender keine Kenntnisse über Interna der verwendeten Speichermechanismen haben muss. Es wird Entwicklungszeit gespart [Roos02, S.8], so dass man sich ganz auf die Anwendungslogik konzentrieren kann.

- Es besteht die Möglichkeit, JDO in Applikationsserver zu integrieren [siehe Punkt eingebettete Umgebung im Abschnitt 2.2.1.1].

- Der Anwender braucht keine SQL-Kenntnisse zu haben, da der Datenbankzugriff transparent für ihn geschieht.

- Ohne den Programmcode zu verändern, kann jederzeit das von der Applikation verwendete Speichermedium zur Datenhaltung gewechselt werden, da sich kein datenbankspezifischer Code im Quellcode befindet.

- JDO übernimmt automatisch die Abbildung des Objektmodells auf das verwendete Speichermedium, so dass der Entwickler alle Vorteile der objektorientierten Programmierung (Vererbung, Komposition, usw.) nutzen kann und sich nicht wie bei den relationalen Datenbanken eigene Gedanken über diese Abbildung machen muss [siehe Nachteile RDBMS Abschnitt 2.1.2]

[1] Die Basiseigenschaften und optionalen Eigenschaften von JDO werden im Abschnitt 2.2.4.2 erläutert

[2] Die Evaluierungskriterien sind in Abschnitt 2.5 zu finden

[3] Der Begriff Transparenz bzw. transparente Persistenz wird im Abschnitt 2.2.4.1 näher erläutert

- Java basierte Abfragesprache JDOQL[1]

2.2.1JDO-Architektur

In diesem Abschnitt wird die JDO-Architektur näher erläutert bzw. auf die Umgebungen, in denen JDO lauffähig ist, sowie auf grundlegende Begriffe eingegangen. Eine JDO-Implementierung ist in einer eingebetteten (managed) und einer eigenständigen (non-managed) Umgebung lauffähig. Im folgenden werden beide Umgebungen vorgestellt:

2.2.1.1Umgebungen von JDO

eigenständige Umgebung

In einer eigenständigen Umgebung laufen Applikationen, die als alleinstehende oder als Client-Server- Architektur implementiert wurden.

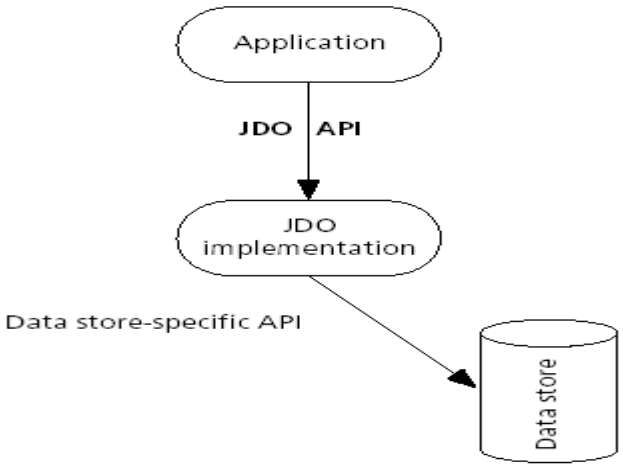


Abbildung 1 „JDO in einer eigenständigen Umgebung [Roos02]“

Hierzu nutzt der Anwender zwei wichtige Schnittstellen der JDO-API, um mit der JDO-Implementierung zu kommunizieren. Die wichtigste Schnittstelle ist der `PersistenceManager`. Diese muss von jedem JDO-Anbieter implementiert werden. Der `PersistenceManager` dient als Schnittstelle zur persistenten Speicherung von Objekten aus der Applikation. Des weiteren stellt er ein Transaktionskonzept [siehe Abschnitt 2.2.5] nach dem ACID-Prinzip[2] zur Verfügung und bietet die Möglichkeit zur Definition von Anfragen[3] an die Datenbank. Die durch den `PersistenceManager` verwalteten Objekte müssen die Schnittstelle `PersistenceCapable`[4] implementieren. Dies kann explizit durch den Programmierer oder implizit durch den Erweiterungsvorgang (Enhancement[5]) erfolgen. Die Schnittstelle

[1] Java Data Objects Query Language: diese wird im Abschnitt 2.2.6

[2] ACID: Eine Transaktion kann durch die Eigenschaften Atomarität (Ausführung als Ganzes), Konsistenz (in sich konsistent), Isolation (isoliert von anderen Transaktionen) und Dauerhaftigkeit (nach erfolgreicher Durchführung dauerhaft gespeichert) beschrieben werden

[3] Es existiert eine spezielle Anfragesprache in JDO names JDOQL, die im Abschnitt 2.2.6 näher erläutert wird

[4] `persistence-capable` sagt nur aus, dass die Option da ist, um dieses Objekt persistent zu speichern. Ob dies geschieht oder nicht, obliegt immer noch dem Programmierer bzw. der Applikation durch expliziten Aufruf der Methode `makePersistent(Object obj)` [Erklärung der Methode Abschnitt 2.2.2]

`PersistenceCapable` deklariert ein Objekt als JDO-Exemplar und stellt Funktionalität zur Erzeugung von persistenten Objekten zur Verfügung. Die JDO-Implementierung kommuniziert dann transparent für den Anwender mit dem verwendeten Speichermedium. Die Evaluierung wird in dieser Umgebung durchgeführt.

eingebettete Umgebung

In dieser Umgebung ist JDO in einen J2EE Applikationsserver integriert. Dies ist eine robuste, komponenten-basierte Umgebung, die immer wieder benötigte Funktionalitäten bereitstellt bzw. bei Installation der eigenen Anwendung den entsprechenden Programmcode für Transaktions-Management, Sicherheit, Nebenläufigkeit und Persistenz bereitstellt. [Oech01]

Da diese Umgebung für die Diplomarbeit nicht in Frage kommt, wird für den interessierten Leser auf die Spezifikation [Russ02] verwiesen, um ausführlichere Informationen zu erhalten.

2.2.1.2 Erweiterungsvorgang (Enhancement)

Die meisten JDO-Implementierungen werden mit einem Enhancement-Tool ausgeliefert. Nach dem die erstellten Klassen mit einem Standard-Java-Compiler übersetzt worden sind, kann das Tool angewendet werden. In einer zuvor erstellen XML-Beschreibungsdatei, dem sogenannten Persistenz-Deskriptor [siehe Abschnitt 2.2.3], werden die zu speichernden Klassen bzw. deren Attribute definiert. Während dem Enhancement-Vorgang [siehe 2.2.1.2] wird dieser Persistenz-Deskriptor ausgewertet, und die geschriebenen Klassen werden mit JDO-spezifischen Methoden am Bytecode erweitert, so dass jede Änderung der Daten eines persistenten Objektes zur Laufzeit erkannt und gegebenenfalls mit der zugrunde liegenden Datenbank synchronisiert werden kann.

[5] Enhancement: dieser Vorgang übernimmt die automatische Abbildung des Java-Objektes auf die Felder der zugrundeliegenden Datenbank

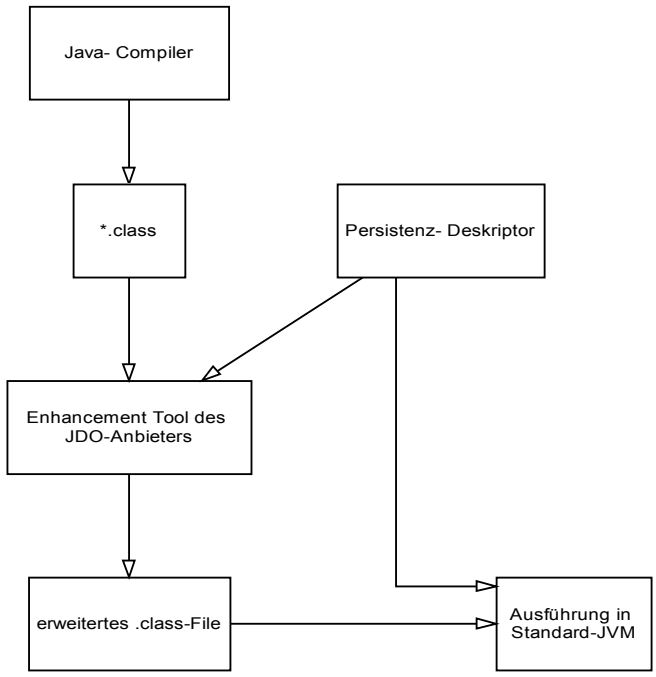


Abbildung 2„Enhancement-Vorgang“

Die XML-Beschreibungsdatei muss zur Laufzeit im CLASSPATH angegeben werden, damit bestimmte Abbildungseigenschaften zur Laufzeit berücksichtigt werden können. Die Verwendung dieses Werkzeuges bietet dem Anwender 100%ige Transparenz und schützt ihn vor manuell erzeugten Fehlern.

2.2.1.3Grundlegende Begriffe

Im folgenden werden grundlegende Begriffe von JDO erläutert:

transiente Objekte:

Ein Objekt ist transient, wenn es erstmals durch den `new()`−Operator erzeugt wurde. Diese Instanzen benötigen keine Persistenz-Infrastruktur, da sie keine persistenten Daten aus der Datenbank repräsentieren, d.h.: ein `PersistenceManager` ist nicht nötig, lediglich der Verweis[1] auf die JDO-Schnittstellen muss im CLASSPATH gesetzt sein. Änderungen an transienten Objekten wirken sich nicht an den Werten in der Datenbank aus, es sei denn der Zustand dieses Objektes wird durch Aufruf der Methode `makePersistent(...)` persistent. An dieser Stelle wird der Begriff „Persistenz durch Erreichbarkeit“ eingeführt, ein weiterer Vorteil von JDO. Damit ist gemeint, dass wenn ein persistentes Objekt ein oder mehrere transiente Objekte referenziert, diese transienten Objekte automatisch, d.h. ohne expliziten Aufruf der Methode `makePersistence(...)`, den Zustand persistent erhalten, und somit als

[1] Dies ist wichtig, da durch expliziten Aufruf der Methode `makePersistence(...)` der Zustand dieses Objektes von transient nach persistent wechseln kann

second-class[1] Objekte mit abgespeichert werden. Ein transientes Objekt wird durch den Garbage-Collector[2] gelöscht, sobald die JVM beendet wird.

persistente Objekte:

Die Erzeugung von persistenten Objekten geschieht auch durch Aufruf des `new()`-Operators. Da diese Objekte die Schnittstelle `PersistenceCapable` implementieren, weiß man, dass sie persistente Daten aus der Datenbank repräsentieren. Diese Objekte sind an einen `PersistenceManager` gebunden, daher werden Änderungen an diesen automatisch innerhalb von einer Transaktion mit der Datenbank synchronisiert. Nach Beendigung der JVM bleibt der Zustand dieser Objekte erhalten, da sie in der Datenbank abgespeichert sind, und somit jederzeit wiederhergestellt werden können.

transaktional:

Der Begriff transaktional sagt aus, dass Objekte innerhalb einer Transaktion verwaltet werden. Jede JDO-Implementierung unterstützt Transaktionen. Die Transaktionskonzepte von JDO werden in Abschnitt 2.2.5 näher erläutert. Die JDO-Implementierung bietet hierzu ein Cache-Mechanismus an. Wird ein Objekt erstmalig in eine Transaktion einbezogen, so werden dessen persistenten Attributwerte durch die JDO-Implementierung zwischengespeichert, so das der Ausgangszustand des Objektes gesichert ist. Nun kann mit diesem Objekt gearbeitet bzw. Veränderungen an diesem durchgeführt werden. Nach einem erfolgreichem commit-Vorgang[3], werden diese gecachten Werte verworfen. Treten während der Transaktion Fehler auf, so wird eine Ausnahme (Exception) geworfen, und durch einen Rollback-Vorgang wird der alte Zustand des Objektes, durch Auslesen der zwischengespeicherten Werte, wiederhergestellt.

nicht-transaktional:

Eine nicht-transaktionale Verwaltung sagt aus, das die Zustände der Objekte nicht in dem Cache der JDO-Implementierung zwischengespeichert werden, und somit nach Änderungen nicht wiederhergestellt werden können.

Typischerweise werden transiente Objekte nicht-transaktional und persistente Objekte transaktional durch JDO verwaltet. Diese beiden Arten der Verwaltung muss von jedem JDO-Anbieter implementiert werden, so dass eine standard-konforme Implementierung vorliegt. Zwei weitere Arten der Verwaltung schreibt die Spezifikation als optionale Eigenschaft von JDO aus: transient-transaktional und persistent-nicht-transaktional. Detailiertere Informationen über diese beiden Verwaltungsarten sind in der Spezifikation von JDO [Russ02] zu finden.

[1] Der Begriff „second-class-Objekt“ wird im Abschnitt 2.2.4.3 eingeführt

[2] Ist die automatische Speicherverwaltung von Java

[3] Durch Aufruf der Methode `commit()` der Schnittstelle `Transaction` wird eine Transaktion beendet, und die Änderungen an diesem Objekt mit der Datenbank synchronisiert

2.2.1.4Das JDO-Identitätenkonzept

Objekt-Identität ist notwendig, um bestimmte Instanzen aus der Datenbank zu suchen/laden bzw. die Eindeutigkeit der Objekte in der JVM zu garantieren. Java unterstützt zwei Arten von Identität: equality und equivalence.

Equality wird durch den `==`-Operator eingeleitet, und liefert true, wenn zwei Referenzen auf das gleiche Objekt im Speicher verweisen. Equivalence vergleicht anhand der Attributwerte zwei Objekte und wird durch die `equals()`–Methode eingeleitet.

Um die Transparenz von JDO auf existierende Objekt-Modelle aufrecht zu erhalten, d.h. unter Verwendung von JDO die existierenden Objekt-Modelle nicht verändern zu müssen, definiert die Spezifikation ein eigenes Identitätenkonzept. Es werden drei Arten von Identität: 1. die Datenbank verwaltet die Identität der Objekte automatisch (Datenbank-Identität), 2. ein durch den Entwickler explizit programmiertes Identitätskonzept (Applikation-Identität) und 3. ein kurzlebiges Identitätskonzept, definiert. Im Persistenz-Deskriptor wird angegeben, welches Konzept für die jeweilige Klasse bzw. deren Objekte, verwendet wird.

Die JDO-Implementierung ist verantwortlich dafür, dass immer nur eine JDO-Instanz[1] mit einem Datenbank-Objekt pro PersistenceManager assoziiert ist. Das Objekt, dass die Instanzen-Identität einkapselt, wird Objekt-ID und die zugehörige Klassendefinition Primary-Key-Klasse[2] genannt. Im folgenden werden die drei Identitätskonzepte vorgestellt. Eine detailliertere Beschreibung ist in der Spezifikation [Russ02] zu finden.

Datenbank-Identität

Dies ist der voreingestellte JDO-Identitäts-Mechanismus, und muss von jedem JDO-Anbieter implementiert werden.

Die Identität wird einem Objekt in diesem Modus automatisch von der Datenbank zugewiesen, wenn das Objekt den Zustand persistent erlangt. Die Art wie dies geschieht, ist von der JDO-Implementierung und der zugrunde liegenden Datenbank abhängig und geschieht transparent für den Anwender. Ist die Identität einmal festgelegt, kann dadurch das Objekt in der Datenbank gefunden und ausgelesen werden[3].

Datenbank-Identität wird im Persistenz-Deskriptor für eine bestimmte Java-Klasse wie folgt eingeleitet:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jdo SYSTEM file:///Verweis auf den Pfad>
<jdo>
        <package name="Verweis auf Modul">
        <class name="Klassenname" identity-type="datastore" />
        </package>
```

[1] Im folgenden ist mit JDO-Instanz ein Objekt gemeint, dass die Schnittstelle `PersistenceCapable` und deren Methoden implementiert hat

[2] Primärschlüsselklasse: In dieser Klasse werden die Attribute spezifiziert, die die Spalte(n) in der Tabelle deklarieren, um eine Zeile aus dieser eindeutig identifizieren zu können, d.h. der Wert muss eindeutig und von Null verschieden sein

[3] Die JDO-Spezifikation bietet dazu mehrere Methoden an, die im Abschnitt 2.2.2 kurz vorgestellt werden

```
</jdo>
```

Applikation-Identität

Bei diesem Modus ist der Anwender verantwortlich für die Objekt-Identität, d.h.: er schreibt eine Primary-Key-Klasse. Durch den Persistenz-Deskriptor werden bestimmte Attributwerte als Primärschlüssel bezeichnet, des weiteren wird die Primary-Key-Klasse namentlich bekannt gegeben. Diese Art der Identität muss von jedem JDO-Anbieter implementiert werden.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jdo SYSTEM file:///Verweis auf den Pfad>
<jdo>
        <package name="Verweis auf Modul">
        <class name="Klassenname" identity-type="applikation"
                objectid-class ="Primary-key-Klasse">
        <field name ="Attribute werden angegeben" primary-key="true" />
        </package>
</jdo>
```

Folgende Eigenschaften muss die Primary-Key-Klasse erfüllen[1]:

- Es muss eine öffentliche Klasse sein, die die Schnittstelle `java.io.Serializable` implementiert
- Die Klasse muss einen argumentlosen Konstruktor und einen Konstruktor mit einem `String`-Argument zur Verfügung stellen
- Die `toString()`-Methode muss überschrieben werden. Diese muss einen `String` als Rückgabewert liefern, so dass dadurch als Übergabe an den `String`-Konstruktor ein äquivalentes Objekt der Primary-Key-Klasse erstellt werden kann
- alle nicht-statischen Attributwerte müssen einen primitiven[2] Datentyp besitzen, oder zu einer anderen öffentlichen Klasse, die die Schnittstelle `java.io.Serializable` implementiert, referenzieren
- für jeden Attributwert, der durch den Persistenz-Deskriptor als Bestandteil des Primärschlüssel definiert wurde, muss ein namentlich gleicher Attributwert in der Primary-Key-Klasse vorliegen, der durch die Methoden `equals()` und `hashcode()` für die Äquivalenz verwendet wird

kurzlebige Identität

Dieser Modus wird verwendet, wenn es bedeutungslos ist, ein Objekt von einem anderen zu unterscheiden, meist bei kurzlebiger Persistenz von neuen Instanzen; z.Bsp.: Alarm-Meldungen vom System. Im Persistenz-Deskriptor, wird dieser Modus im Tag `<package>` unter `identity="nondurable"` aktiviert. Da dies eine optionale

[1] Die gleichen Eigenschaften, wie bei einem Remote-Objekt von RMI. Dadurch wird erreicht, dass diese Primary-Key-Klasse in einer verwalteten Umgebung auch als EJB-Primary-Key-Klasse verwendet werden kann

[2] Damit sind folgende Datentypen gemeint: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float` und `double`

Eigenschaft von JDO ist, wird der interessierte Leser auf die Spezifikation [Russ02] verwiesen.

2.2.2Klassen, Interfaces und Packages

Die JDO-Spezifikation besteht aus 2 Packages: `jdox.jdo.spi` und `jdox.jdo`.

Das Paket `jdox.jdo.spi` beinhaltet alle Klassen und Schnittstellen, die intern von der JDO-Implementierung verwendet werden. Da der Anwender darauf keinen Zugriff hat, wird der interessierte Leser auf [Russ02] verwiesen.

Im Paket `jdox.jdo` befinden sich Klassen, Schnittstellen und Exceptions[1] mit denen der Anwender konfrontiert wird, d.h. die dort definierten Schnittstellen werden von dem jeweiligen JDO-Anbieter implementiert und stehen durch die der JDO-Implementierung mitgelieferten API dem Anwender zur Verfügung. Dieser Abschnitt gibt einen Überblick auf die wichtigsten Schnittstellen und Klassen des Paketes `jdox.jdo`:

- `PersistenceManagerFactory`
  Über Parameter (Datenbankkonfigurationen, Transaktionskonzept, usw. siehe [Russ02, S. 75ff] wird diese Schnittstelle konfiguriert. Sie dient als Quelle für 1...n `PersistenceManager`.
- `PersistenceManager`
  Instanzen dieser Schnittstelle kapseln die JDBC-Verbindung zu der verwendeten

  Datenbank. Über einen `PersistenceManager` können `Query`- und `Transaction`- Instanzen bezogen werden.
- `Query`
  Über eine Query können Abfragen durch JDOQL[2] an die verwendete Datenbank geschickt werden.
- `Transaction`
  Diese Schnittstelle definiert Methoden für das Transaktionskonzept.

- `InstanceCallbacks`
  Dort wird der Callback-Mechnismus definiert (dieser wird in Abschnitt 2.2.4.5 näher erläutert).
- `Extent`
  Ein Extent repräsentiert eine Sammlung aller Instanzen einer bestimmten Klasse aus der Datenbank.
- `JDOHelper`
  Statische Methoden werden in dieser Klasse zur Verfügung gestellt, über die die Zustände[3] der JDO-Instanzen abgefragt werden können.

[1] In Abschnitt 2.2.7 wird die Ausnahmebehandlung von JDO erläutert

[2] Die JDO-Spezifikation definiert eine eigenen Abfragesprache names JDOQL. Diese wird in Abschnitt 2.2.6 erläutert.

[3] In Abschnitt 2.2.4.4 werden die möglichen Zustände der JDO-Instanzen anhand eines Zustandsdiagramm vorgestellt.

2.2.3Persistenz-Deskriptor

Der Persistenz-Deskriptor wird während des Enhancement-Vorgangs ausgewertet, um zu erfahren, welche Klassen bzw. Attribute persistent in der Datenbank abgelegt werden sollen bzw. während der Applikation bei Änderungen an diesen mit der Datenbank synchronisiert werden. Des weiteren wird er von manchen JDO-Anbieter-Implementierungen während der Laufzeit der Applikation ausgewertet, um bestimmte Mapping-Informationen zu erhalten z.B.: wird festgelegt, welches Attribut einer Klasse welchem Feld einer Tabelle entspricht, und welche Beziehungen zwischen den Klassen und Tabellen besteht. Dazu steht das Tag `<extension>` für anbieterspezifische Angaben zur Verfügung.

Die Beschreibung des Persistenz-Deskriptors erfolgt in Form eines XML-Dokumentes, welches mit einer Grammatik (DTD-File[1]) übereinstimmen muss. [siehe Abschnitt 6.14]

Der erstellte Persistenz-Deskriptor muss auf dieses DTD-File jdo.dtd verweisen.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jdo SYSTEM file:/// entsprechende Pfad-Angabe>
<jdo>
…
</jdo>
```

Nach Spezifikation muss der Persistenz-Deskriptor folgende Namenskonvention erfüllen:

„Klassen-oder Package-Name" . jdo

Eine detaillierte Erläuterung der einzelnen Tags ist in **[Russ02]** zu finden.

2.2.4Objekt-Modell

Dieser Abschnitt beschreibt die unterstützen Typen und die Konfiguration der Objekte von JDO, des weiteren wie diese durch den PersistenceManager verwaltet werden, untereinander interagieren und die Zustände der JDO-Instanzen.

2.2.4.1Transparente Persistenz

Während des Ablaufes einer Applikation beinhaltet die Java Virtual Machine transiente Objekte. Manche von diesen sind normale (nicht peristence-capable) Objekte, andere sind Instanzen von oder referenzieren auf persistence-capable-Klassen. Jeder `PersistenceManager` verwaltet einen Cache von Instanzen, der durch JDO gefüllt wird, sobald eine Instanz durch die Applikation referenziert wird.

[1] Document Type Definition: Dort wird die Struktur eines XML-Elementes definiert. Die DTD beschreibt die unterstützten Elemente, deren Attribute, sowie die Elemente welche innerhalb des Dokumentes in anderen Elementen auftreten dürfen.

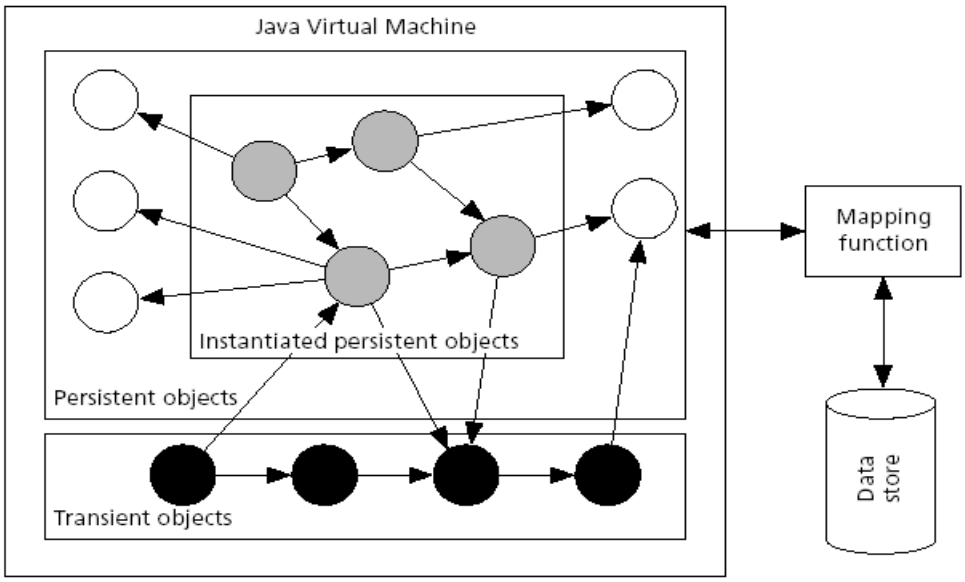


Abbildung 3„JDO-Transparenz [Roos02]“

Die Applikation teilt der JDO-Implementierung mit, welche Objekte aus der Datenbank gelesen, oder in die Datenbank abgespeichert werden sollen. Transparent für den Anwender liest dann die JDO-Implementierung diese Objete aus der Datenbank aus, oder speichert sie in der Datenbank ab. Da persistente Objekte in einer Datenbank gespeichert, oder aus dieser geladen werden, existiert eine Mapping-Funktion zwischen dem eigentlichen Objekt und der Abbildung in der Datenbank. Diese Mapping-Funktion wird nicht durch den JDO-Standard spezifiziert, sondern unterliegt dem Entwickler.

Transparente Persistenz wird nach [Roos02] wie folgt beschrieben:

- die Illusion, dass alle persistenten Objekte im Speicher sofort verfügbar sind, und nicht erst aus der Datenbank gelesen werden müssen
- dem impliziten Update von dirty (die verschiedenen JDO-Zustände werden weiter unten erklärt) persistenten Instanzen mit der Datenbank durch ein commit
- das automatische Mapping von Java-Datentypen auf die Datentypen der zugrunde liegenden Datenbank

2.2.4.2Unterstützte Datentypen und Modifikatoren von JDO

Eine JDO-Instanz kann persistente und nicht-persistente (werden nicht durch die JDO-Infrastruktur verwaltet) Attributwerte beinhalten. Persistente Attributwerte unterstützen alle gültigen Kombinationen von Java-Modifikatoren: `private`, `public`, `protected`, `static`, `transient`, `abstract`, `final`, `synchronized` und `volatile`. Aber nicht alle Java-Datentypen werden von der JDO-Spezifikation als benötigt vorausgesetzt, sondern können von den JDO-Anbieter als optionale Eigenschaft implementiert werden. Die Wahl der zu untersuchenden JDO-Implementierungen für diese Diplomarbeit ist davon und von den unterstützenden Datenbanken, abhängig [siehe Abschnitt 2.3].

Objekte von System-Klassen, die laufzeitabhängig sind z.B. `System`, `Thread`, `Runtime` des Packages `java.lang`, Klassen des Packages `java.io` und `java.net` können nicht persistent abgelegt werden. Der Grund ist, dass Objekte dieser Klassen aktuelle laufzeitspezifische bzw. intern plattformspezifische Informationen beinhalten, und diese nicht wieder rekonstruiert werden können.

Die JDO-Spezifikation schreibt eine Reihe von zu unterstützenden Java-Datentypen voraus, die von jedem JDO-Anbieter JDO-spezifisch implementiert werden müssen, so dass diese Implementierung standard-konform ist. Es folgt eine Auflistung dieser Datentypen:

Persistente Werte

Es folgt eine Datentypauflistung, die von jeder JDO-Implementierung unterstützt werden müssen:

- alle primitiven Java-Datentypen: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` und `double`
- Wrapper-Klassen: `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `String`, `Locale`, `BigDecimal` und `BigInteger`
- Die beiden Schnittstellen `Collection` und `Set`. Konkret muss aber nur die Klasse `HashSet` und `Date` von dem JDO-Anbieter implementiert werden.
- eigene Datentypen, die die Schnittstelle `PersistenceCapable` implementieren
- Der Datentyp `Object`. Wird während der Implementierung dieser Datentyp verwendet, so muss der Entwickler eine explizite Typanpassung vornehmen, da in diesem Datentyp alle Arten von Objekten gespeichert werden können
- eigene Interface-Definitionen

Weiterhin definiert die JDO-Spezifikation optionale Datentypen:

- Schnittstellen `Map` und `List` mit folgenden konkreten Implementierungen: `ArrayList`, `HashMap`, `Hashtable`, `LinkedList`, `TreeMap`, `TreeSet` und `Vector`

Der Grund, dass diese konkreten Klassenimplementierungen als optionale Datentypen spezifiziert wurden liegt daran, dass JDO auch durch die Java Mirco Edition[1] unterstützt wird. Diese Entwicklungsumgebung unterstützt aber nicht die Schnittstellenhierarchie `Collection`. Auch hiervon ist die Wahl der zu evaluierenden JDO-Implementierung abhängig. [siehe Abschnitt 2.3]

- Unterstützung von Array-Objekten, sowie die Benachrichtigung von deren Änderungen an die besitzenden first-class-Objekte[2]

[1] Java Micro Edition: Entwicklungsumgebung, mit der mobile Applikationen entwickelt werden könnnen

[2] Der Begriff first-class-Objekt wird auf der nächsten Seite eingeführt

2.2.4.3Vererbung unter JDO

Die Vererbung wird von JDO unterstützt. Eine Klasse kann persistence-capable sein, aber deren Superklasse muss es nicht sein, d.h.: es existiert eine Unabhängigkeit zwischen persistence-capable und nicht-persistence-capable Klassen. Wird eine Klasse in einer Vererbungshierarchie als persistence-capable definiert, so werden die Attribute, die durch die Persistenz-Modifizierer: `persistent`, `transactional` oder `none` beschrieben werden, an die Unterklasse vererbt, und liegen dort als solche vor. Wenn eine persistence-capable-Klasse eine nicht-persistence-capable-Oberklasse hat, so werden die Attribute der Oberklasse nicht durch JDO[1] mit der Datenbank synchronisiert. Falls dies doch gewünscht wird muss versucht werden, die Oberklasse als persistence-capable zu definieren, oder aber diese Werte, welche persistent abgelegt werden sollen, in der Unterklasse zu überschreiben. Für deren Veränderungen bzw. die Weitergabe der Veränderungen an die Oberklasse ist der Benutzer selbst verantwortlich.

Wird in einer Vererbungshierarchie eine Klasse als abstrakt definiert, so wird auch, falls Applikation-Identifikation verwendet wurde, die zugehörige Primary-Key-Klasse abstrakt, d.h.: alle von dieser Klasse erbenden persistence-capable Unterklassen müssen eigene Implementierungen für die Applikation-Identifikation bereit stellen, welche von der abstrakten Ober-Primary-Key-Klasse erben.

Im folgenden werden die Begriffe first-class- und second-class-Objekte eingeführt:

First-class-Objekte

Dies sind Instanzen von PersistenceCapable-Klassen, welche eine JDO-Identität (siehe Abschnitt 2.2.1.4) haben. Dadurch liegen sie als eindeutige Objekte im Cache des PersistenceManager vor. Wird ein Attributwert eines first-class-Objektes verändert, so wechselt sein Zustand in den Zustand dirty[2].

Second-class-Objekte

Diese Objekte implementieren nicht die Schnittstellte `PersistenceCapable` und besitzen keine JDO-Identität, d.h. sie liegen nicht eindeutig im Cache des PersistenceManager vor. Trotzdem wird eine gewisse Persistenz-Unterstützung durch die JDO-Implementierung zur Verfügung gestellt, weil sie als Bestandteil eines first-class-Objektes abgespeichert werden (Komposition[3]). Wird ein second-class-Objekt verändert, wechselt nicht dieses, sondern das es besitzende first-class-Objekt in den Zustand dirty. Will man auf ein second-class-Objekt aus der Datenbank zugreifen, kann dies nicht direkt durch die Objekt-ID geschehen, da keine JDO-Identität vorliegt,

[1] Wird nicht durch die Version 1.0 unterstützt. Steht aber als Spezifikationserweiterung JDO Version 1.0.1 als umzusetzende Eigenschaft an

[2] Die verschiedenen Zustände bzw. die Zustandsübergänge, die eine JDO-Instanz während ihrer Lebensdauer einnehmen kann, werden auf der nächsten Seite anhand eines Zustandsdiagramm erläutert

[3] Komposition: Zwischen den Objekten liegt eine Beziehung vor, die sich als „ist Teil von“ oder „besteht aus“ beschreiben lässt. Beim Löschen des Ganzen (first-class-Objekt) müssen auch alle Teile (second-class-Objekt) gelöscht werden. [Balz99]

sondern man greift auf das es besitzende first-class-Objekt zu und navigiert durch dessen persistenten Attribute, und erreicht somit das second-class-Objekt.

Im folgenden werden die Zustände der JDO-Instanzen während ihrer Lebensdauer, d.h. bis sie von der automatischen Speicherverwaltung gelöscht werden, erläutert:

2.2.4.4Zustände der JDO-Instanzen

Transient

Der Zustand transient wurde schon in Abschnitt 2.2.1.3 eingeführt.

Persistent-New

Ein Objekt wechselt vom transienten in den Zustand new, sobald dieses während der aktuellen und noch nicht abgeschlossenen Transaktion durch Aufruf der Methode `makePersistent(...)` gespeichert wird. Der PersistenceManager weist dieser Instanz dann automatisch eine JDO-Identität zu.

Persistent-New-Deleted

Der Zustand new-deleted wird erreicht, wenn ein ein transientes Objekt während einer Transaktion zum Zustand persistent-new wechselt, und anschließend diesem Objekt in der aktuellen und noch nicht beendeten Transaktion durch Aufruf der Methode `deletePersistence(...)` der Zustand persistent-new entzogen wird.

Hollow

Der Zustand hollow sagt aus, dass ein Objekt mit der Datenbank in Beziehung steht (referenziert ist). Dies ist eine Art Zwischenzustand, der erreicht wird, wenn u.a. eine Instanz ihre JDO-Identität geladen hat, d.h.: die als Primary-Key definierten Attributwerte, aber noch nicht die restlichen persistenten Werte. Dieser Zustand garantiert die Einzigartigkeit von Instanzen zwischen Transaktionen.

Persistent-Clean

Eine Instanz befindet sich in diesem Zustand, wenn in der aktuellen Transaktion die Werte aus der Datenbank geladen, aber nicht geändert wurden. Dieser Zustand definiert einen konsistenten Zustand, da die Daten im repräsentierenden Objekt identisch mit denen in der Datenbank sind.

Persistent-Dirty

Dieser Zustand wird erreicht, wenn sich die Attributwerte des Objektes in der aktuellen Transaktion geändert haben, d.h. es liegt ein inkonsistender Zustand zwischen den Werten im Objekt und der Datenbank vor.

Persistent-Deleted

Wenn ein schon persistentes Objekt in der aktuellen Transaktion gelöscht wird.

Um das Erreichen der jeweiligen Zustände näher zu erläutern, folgt ein Zustandsdiagramm:

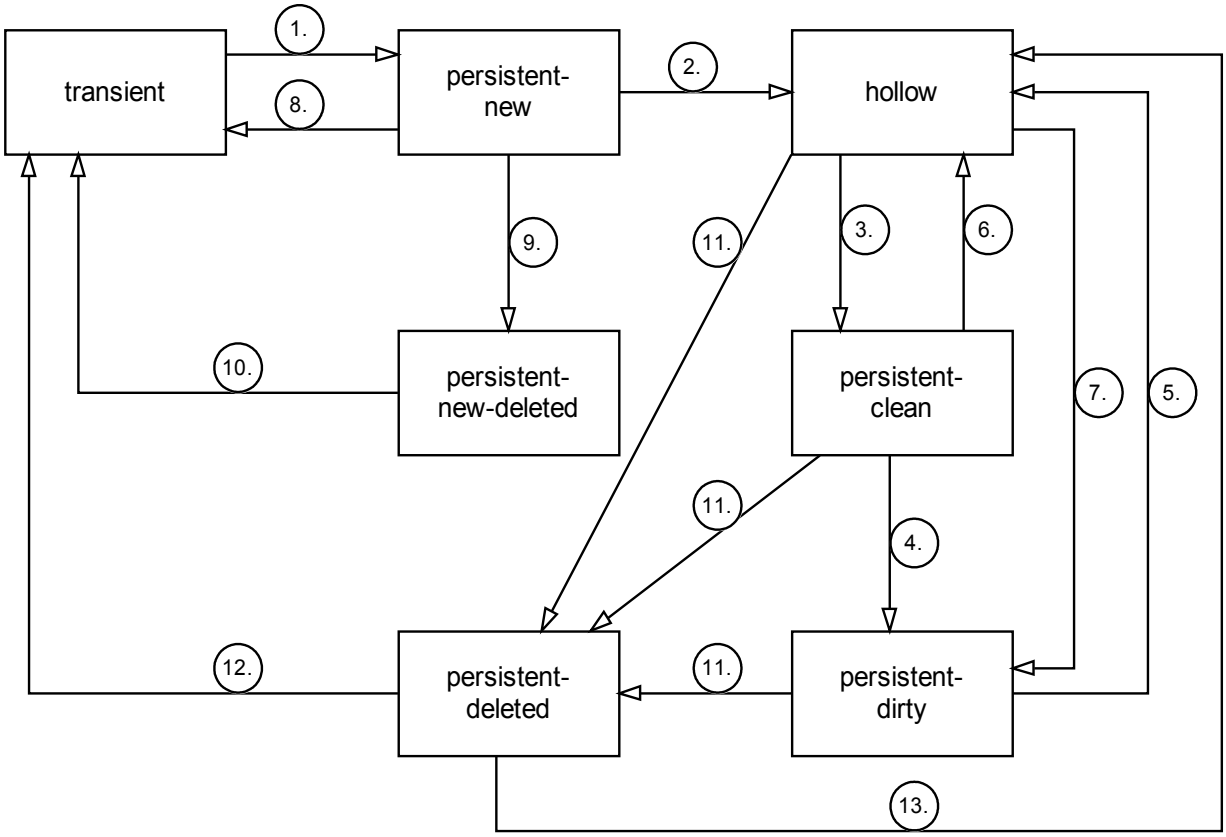


Abbildung 4„Zustandsdiagramm der nach JDO-Spezifikation geforderten sieben Zustände“

1. Ein transientes Objekt wechselt in den Zustand persistent-new, wenn dieses als Argument an die Methode `makePersistence(...)` übergeben wurde.

2. Das zuvor persistente Objekt wechselt in den Zustand hollow, wenn die aktuelle Transaktion erfolgreich beendet wurde (nach dem Commit-Vorgang).

3. Ein Zustandsübergang von hollow nach persistent-clean findet statt, wenn Daten aus der Datenbank eingelesen werden.

4. Der Zustand dirty wird erreicht, sobald Attributwerte des Objektes geändert werden.

5. Nach Änderungen der Attributwerte des Objektes wechselt dieses vom Zustand dirty in den Zustand hollow, wenn die aktuelle Transaktion abgebrochen (Rollback-Vorgang) oder erfolgreich beendet wurde (Commit-Vorgang).

6. Wurden die Attributwerte des Objektes nach dem Einlesevorgang noch nicht geändert, wechselt der Zustand es Objektes von persistent-clean in den Zustand hollow, sobald die aktuelle Transaktion abgebrochen oder erfolgreich beendet wurde.

7. Befindet sich das Objekt in dem Zustand hollow, so wechselt sein Zustand von hollow in den Zustand persistent-dirty, sobald die zurvor eingelesenen Attributwerte verändert wurden.

8. Ein zuvor persistentes Objekt wechselt wieder von dem Zustand persistent-new in den Zustand transient, sobald die aktuelle Transaktion abgebrochen wurde.

9. Als Argument der Methode `deletePersistence(...)` wird dem persistentem Objekt der Zustand persistent wieder entzogen und wechselt somit in den Zustand persistent-new-deleted.

10. Wird die aktuelle Transaktion abgebrochen oder erfolgreich beendet, so wechselt das Objekt vom Zustand persistence-new-deleted in den Zustand transient.

11. Als Argument der Methode `deletePersistence(...)` wechselt das Objekt von den Zuständen hollow, persistent-clean und persistent-dirty in den Zustand persistent-deleted.

12. Wenn die aktuelle Transaktion, indem dem Objekt der Zustand persistent entzogen wurde, erfolgreich beendet, so wechselt sein Zustand von persistent-deleted in den Zustand transient.

13. Wenn die aktuelle Transaktion, indem dem Objekt der Zustand persistent entzogen wurde, abgebrochen wird, so wechselt sein Zustand von persistent-deleted in den Zustand hollow.

Manche Zustände der JDO-Instanzen sind für die Applikation verdeckt, aber intern kennt der `PersistenceManager` diese Zustände, die durch Methoden der `JDOHelper`-Klasse auch vom Anwender abgefragt werden können.

- `public static boolean isDeleted(Object pc)`
  liefert true, wenn das Object in der aktuellen Transaktion gelöscht wurde
- `public static boolean isDirty(Object pc)`
  liefert true, wenn ein persistenter Attributwert in der aktuellen Transaktion geändert wurde
- `public static boolean isNew(Object pc)`
  liefert true, wenn das Objekt in der aktuellen Transaktion persistent wird
- `public static boolean isPersistent(Object pc)`
  liefert true, wenn das Objekt persistent ist
- `public static boolean isTransactional(Object pc)`
  liefert true, wenn das Objekt transaktional ist

2.2.4.5Instanzen-Callbacks

Mit Instanzen-Callbacks sind Aktionen gemeint, die bei einem Zugangsübergang ausgeführt werden. Dadurch kann sich der Programmierer durch die JDO-Implementierung benachrichtigen lassen, sobald eine Instanz einen gewissen Zustand erreicht hat oder wenn eine Instanz vor einem Zustandsübergang steht.

Dafür stellt die JDO-Spezifikation die `InstanceCallback`-Schnittstelle mit vier Methoden zur Verfügung, die bei Bedarf implementiert werden können. Diese werden im folgenden kurz erläutert:

- `void jdoPostLoad()`
  wird aufgerufen, sobald die persistenten Attributwerte einer Instanz aus der Datenbank geladen wurden. Dies wird meist verwendet, um die nicht-persistenten Werte der Instanz zu initialisieren
- `void jdoPreStore()`
  wird aufgerufen, sobald die persistenten Werte einer persistenten Instanz mit der Datenbank synchronisiert werden
- `void jdoPreClear()`
  wird aufgerufen, sobald eine Instanz in den Zustand hollow wechselt
- `void jdoPreDelete()`
  wird aufgerufen, sobald eine Instanz in den Zustand persistent-deleted oder persistent-new-deleted wechselt. Dies wird verwendet, um bestimmte Lösch- bzw. Cascade[1]-Regeln zu beachten

2.2.5Transaktionen unter JDO

JDO unterstützt folgende Transaktion-Strategien:

2.2.5.1pessimistische Transaktionen (JDO-Standard)

Meist verwendet bei kurzlebigen Transaktionen. Vom Beginn des ersten Datenzugriffs bis zum endgültigen `commit()`, liegt eine aktive Datenbank-Transaktion vor. Wird versucht auf die Daten einer Transaktion lesend oder schreibend zuzugreifen, werden diese solange blockiert, bis die diese Daten benutzende Transaktion, diese wieder freigibt.

Mögliche Verstöße werden erkannt bevor sie auftreten und die auslösenden Datenzugriffe werden verzögert, bis sie ohne Verletzung der Serialisierbarkeit durchgeführt werden können. [Baum94]

Durch die Methode `setOptimistic(false)` wird das pessimistische Transaktionskonzept aktiviert.

2.2.5.2optimistische Transaktionen (optionale Eigenschaft)

Optimistisch meint, dass ein persistentes Objekt während der aktuellen Transaktion inkonsistente Werte gegenüber der Datenbank haben darf. Nach Beendigung der Transaktion (nach commit-Vorgang) werden diese Werte noch einmal mit denen in der Datenbank verglichen. Meist wird dieses Konzept bei langlebigen Transaktionen verwendet, z.B. Dauer, bis der Benutzer während eines Transaktionsvorgangs Daten

[1] Cascade: bedeutet, dass alle Objekte, die von einem Hauptobjekt aus referenziert werden mit gelöscht werden, sobald das Hauptobjekt gelöscht wird

eingetippt hat. Wenn Instanzen aus der Datenbank abgefragt bzw. geladen werden, werden diese erst gesperrt, sobald der Versuch gestartet wird, diese zu ändern bzw. zu löschen. Dazu stehen z.B. das Zeitstempel- oder Versionsverfahren[1] zur Verfügung. Durch die Methode `setOptimistic(true)` wird dieses Konzept aktiviert.

Erst beim Abschluss einer Transaktion wird geprüft, ob sie sich serialisieren lässt. Ist dies unmöglich, wird sie abgebrochen und muss von vorne beginnen. Deshalb durchlaufen optimistische Transaktionen normalerweise drei Phasen: Eine Lesephase, in der Daten gelesen und Berechnungen durchgeführt werden, eine Validierungsphase, in der die Einhaltung der Konsistenzkriterien geprüft wird, und eine Schreibphase, in der die in der Lesephase berechneten Änderungen in die Datenbasis eingetragen werden. [Bau94]

Die Schnittstelle `Transaction` stellt einige Methoden zur Verwaltung von Transaktionen zur Verfügung. Auf die wichtigsten wird hier kurz eingegangen.

Es können in einer Anwendung mehrere PersistenceManager existieren. Zwischen PersistenceManager und Transaktion existiert eine 1...1-Bezieung, d.h. dass ein PersistenceManager immer nur eine aktive Transaktion verwalten kann. Durch die Methode `currentTransaction()` wird dem Anwender eine Instanz der Transaction-Schnittstelle vom JDO-Persistenz-Manager geliefert.

Dieses Transaction-Objekt lebt solange, bis der Persistenz-Manager geschlossen wird, d.h. alle JDO-Transaktionen sind an diesem Objekt gebunden, und werden der Reihe nach ausgeführt. JDO unterstützt nicht das Konzept der verschachtelten Transaktionen.

2.2.5.3Wichtige Methoden der Schnittstelle Transaction

- `boolean isActive()`
  Diese Methode liefert `true`, wenn eine aktive Transaktion vorliegt, d.h.: die `begin()` Methode wurde schon ausgeführt, jedoch noch kein `commit()` oder `rollback()`.
- `void begin()`
  Eine Transaktion wird gestartet.
- `void commit()`
  Benachrichtigung an die unterliegende Datenbank, ihre Transaktion abzuschliesen.
- `void rollback()`
  Vorheriger Ist-Zustand der Daten in der unterliegenden Datenbank wird wiederhergestellt.

JDO definiert zwei weitere Transaktions-Verfahren, um die Effizienz zu verbessern: dies sind die Flags **RestoreValues** und **RetainValues**. Das Setzen dieser Flags wirkt sich nur auf Instanzen im Speicher aus und nimmt keinen Einfluss auf den Ablauf der commit bzw. rollback Prozesse der zugrunde liegenden Datenbank.

RestoreValues

[1] Auf die möglichen Verfahren wird nicht näher eingegangen

Ist dieses Flag auf `true` gesetzt, bedeutet dies, dass nach einem Rollback-Vorgang der alte Zustand wiederhergestellt wird. Diese Informationen werden aus einem Cache gelesen, der bei Beginn der Transaktion mit den alten Werten beschrieben wird.

`False` bedeutet, dass der Zustand des Objektes nach hollow (Erklärung der einzelnen Zustände siehe Abschnitt 2.2.4.4) wechselt, d.h.: der alte Zustand muss nicht wiederherstellt werden, es sei denn die Applikation greift auf diese Daten zu, dann werden die einzelnen Werte aus der Datenbank gelesen und gesetzt. Der Zustand `false` bringt erhebliche Performance-Verbesserungen, falls nach einem Rollback nicht wieder auf die Instanzen zugegriffen wird.

RetainValues

Der Default-Wert ist `false`, d.h. nach einem erfolgreichem commit, wechselt das Objekt in den Zustand hollow. Dies reduziert den Instanzen-Cache und verbessert die Performance.

Der Zustand `true` überlässt dem Entwickler (explizit) oder dem Persistence-Manager (implizit) die Löschung des Instanzen-Caches. Dies bringt Performance-Verbesserungen, falls die Applikation in mehreren unabhängigen Transaktionen mit den gleichen Objekten arbeitet.

Des weiteren kann der Transaktion-Vorgang mit einem callback-Modus versehen werden, so dass die Applikation benachrichtigt wird, falls ein commit-Vorgang eingeleitet wird bzw. abgeschlossen ist. Die Schnittstelle `Synchronization` definiert zwei Methoden `beforeCompletion()` und `afterCompletion(int)` (int-Parameter: `javax.transaction.Status.STATUS_COMMITED` und `STATUS_ROLLEDBACK`), die durch die Methode `setSynchronization( Synchronization sync)` an die Transaktion gebunden werden können.

2.2.6Abfragen mittels JDOQL

Die JDO-Spezifikation definiert eine eigene Abfragesprache namens JDOQL (Java Data Objects Query Language). Anhand von Beispielen gibt dieser Abschnitt dem Leser einen Einblick in JDOQL.

Durch die `Query`-Schnittstelle wird eine objektorientierte und datenbankneutrale Abfrage zur Verfügung gestellt. D.h. die JDO-Implementierung passt die durch die JDOQL-Definition erstellten Abfragen an das entsprechende Speichermedium an, z.B. generiert SQL-Statements über JDBC unter Verwendung von relationalen Datenbanken. Soll die Speicherung in Dateien bzw. XML-Dateien erfolgen, die keine eigene Abfragesprache definieren, so liegt es im Ermessen des JDO-Anbieters, ob er diese Speichermedien unterstützt bzw. wie er die Abbildung von JDOQL auf diese Speichermedien vornimmt.

Durch Filter, die an die `Query` übergeben werden, lässt sich die Abfrage einschränken. Filter stellen einen Java-basierten boolschen Ausdruck dar. Dies wird weiter unten anhand eines Beispieles erläutert. Als Ergebnis liefert die `Query` eine `Collection` von

Instanzen zurück, für die der Filter zu `true` ausgewertet wurde. Weiterhin besteht die Möglichkeit Variablen an die `Query` zu übergeben, welche erst zur Laufzeit ausgewertet werden.

Folgendes Beispiel lädt alle Bücher inklusive Unterklassen aus der Datenbank und stellt sie an der Standardausgabe dar. Die Abfrage ist sehr einfach, da sie keine Filter, Variablen oder Ordnungsreihenfolgen beinhaltet.

```
trans.begin();
Extent bookExt = pManager.getExtent(Book.class, true);
Query query = pManager.newQuery(bookExt);
Collection coll = (Collection) query.execute();
Iterator iter = coll.iterator();
System.out.println("Alle Bücher aus der DB:");
while(iter.hasNext())
{
      Object book = iter.next();
      System.out.println(book);
}
query.close(coll);
trans.commit();
```

Es folgt ein Beispiel unter Verwendung eines Filters und Ausnutzung der Ordnungsreihenfolge. Es werden alle Bücher mit dem Titel: „Java ist auch eine Insel“ aus der Datenbank gelesen. Die Ausgabe erfolgt in aufsteigender Sortierung nach dem Namen und absteigender Sortierung nach der BuchId:

```
trans.begin();
Extent bookExt = pManager.getExtent(Book.class, true);
String filter = „name“ ==\"Java ist auch eine Insel\"";
Query query = pManager.newQuery(bookExt, filter);
query.setOrdering("name ascending, bookId descending");
Collection coll = (Collection) query.execute();
…..
trans.commit();
```

Weitere Möglichkeiten von JDOQL sind unter [Russ02 S.129ff und Roos02 S.100ff] zu finden.

2.2.7Ausnahmebehandlung unter JDO

JDO definiert neun Laufzeit-Exceptions[1]. Diese werden in folgende Kategorien unterteilt:

- fatale Exceptions, d.h.: die Operation kann nicht komplett ausgeführt werden, implizit erfolgt ein rollback der aktuellen Transaktion
- retried Exceptions, d.h.: der Fehler kann durch die Anwendung korrigiert werden, und die Operation wird erneut ausgeführt

Die Exception Hierarchie:

[1] Im folgenden wird der Begriff Exception verwendet. Damit ist ein z.B. Programm-spezifischer oder Datenbank-spezifischer aufgetretener Fehler gemeint

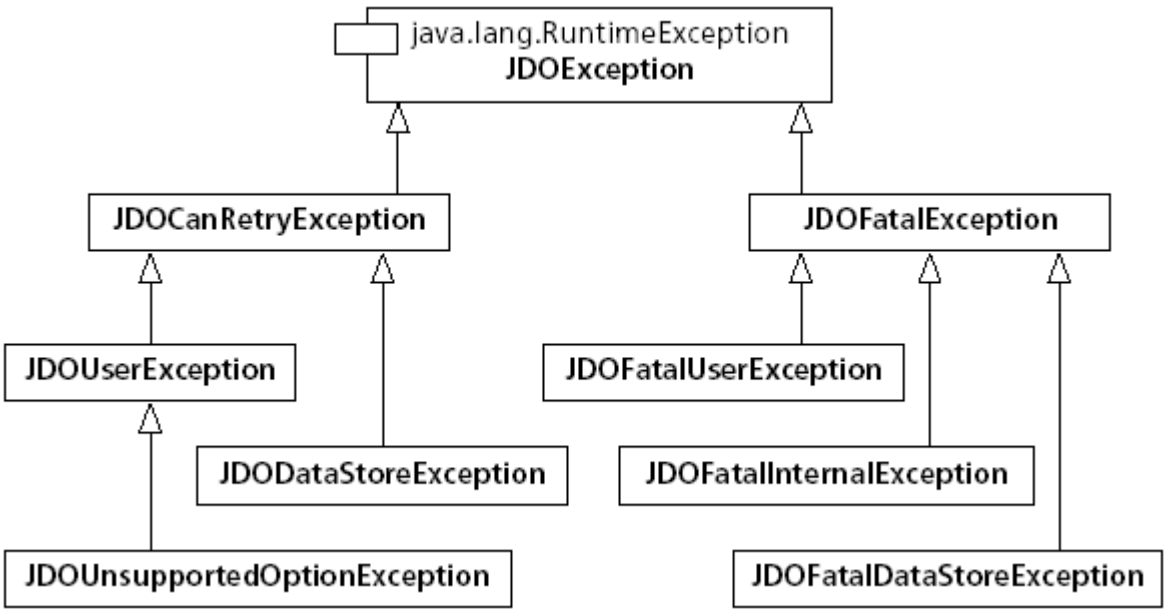


Abbildung 5„Die Exception-Hierarchie [Roos02]“

Der Ausgangspunkt der jeweiligen Exception erkennt man an folgender Terminologie:

- User: ausgelöst durch die Applikation, welche den JDO-Service nutzt
- DataStore: ausgelöst durch die benutzte Datenbank
- Internal: ausgelöst durch die JDO-Implementierung des jeweiligen JDO-Anbieter

Die JDO-Anbieter können eigene Exceptions definieren, die von diesen Standard-Exceptions abgeleitet sind.

Die Basisklasse aller Exceptions

Tritt während der Laufzeit ein Fehler auf, so wird automatisch ein Konstruktor aufgerufen, der diesen Fehler auffängt. Die Argumente können dann ausgelesen werden, so dass man nähere Informationen über den aufgetretenen Fehler erhält.

z.Bsp:

- `JDOException(String msg, Throwable[] nested, Object failed)` `msg`: eine kurze Fehlerbeschreibung, `nested`: eingebettete Exceptions, `failed`: eine Referenz auf das fehlgeschlagene Objekt

Es folgt eine kurze Beschreibung der einzelnen Exceptions:

- `JDOCanRetryException`

Basisklasse für alle Exceptions, die durch die Anwendung wieder rückgängig gemacht werden können.

- `JDOUserException`
  Basisklasse für alle Exceptions, die durch den User verursacht werden.
- `JDOUnsupportedOptionException`
  Wird ausgelöst, wenn die Anwendung auf ein, durch den JDO-Anbieter, nicht implementierte optionale Eigenschaften des JDO-Standard zugreifen möchte.
- `JDODataStoreException`
  Basisklasse für alle Datenbank-spezifischen Exceptions.
- `JDOFatalException`
  Basisklasse für alle fatalen Exceptions, löst implizit ein Rollback der aktuellen Transaktion aus.
- `JDOFatalUserException`
  Basisklasse für alle fatalen Exceptions ausgelöst durch die Anwendung.
- `JDOFatalInternalException`
  Wird ausgelöst, durch interne Fehler der jeweiligen JDO-Implementierung.
- `JDOFatalDataStoreException`
  Basisklasse für alle fatalen Exceptions, die durch die unterliegende Datenbank ausgelöst werden.

2.3 Selektionskriterien der JDO-Implementierungen

In Abschnitt 6.2 folgt eine Auflistung von kommerziellen und open-source JDO-Implementierungen, die zu Beginn der Diplomarbeit durch eine Internetrecherche hinsichtlich diverser Kriterien näher untersucht wurden. Entscheidend für die Selektion waren Antworten auf folgende Fragen:

- Welche Datenbanken werden von der JDO-Implementierung unterstützt?
- Werden weitere über die von der JDO-Spezifikation als benötigt vorgeschriebenen Datentypen unterstützt?
- Welche weiteren optionalen Eigenschaften von JDO wurden implementiert?
- Liegt der untersuchten JDO-Implementierung eine graphische Entwicklungsumgebung bei?
- Liegt der JDO-Implementierung ein Enhancement-Werkzeug und/oder ein Tabellenskript-Generator[1] bei?
- Ist es eine kommerzielle oder open-source JDO-Implementierung?
- Kann die JDO-Implementierung in der eingebetteten Umgebung ausgeführt werden, d.h. wird der Einsatz in einem Applikationsserver unterstützt?

[1] Damit ist ein Werkzeug gemeint, das automatisch aus den zu speichernden Objekten bzw. Klassen die zugehörigen Skripte zum Anlegen der Tabellen erzeugt

Die Ergebnisse der Internetrecherche sind in Abschnitt 6.3 zu finden.

2.4 JDO-Implementierungen (Frameworks)

In diesem Abschnitt werden kurz die untersuchten JDO-Implementierungen vorgestellt, die im Rahmen dieser Diplomarbeit anhand eines Prototypes (Erklärung siehe Abschnitt 3) evaluiert worden sind.

2.4.1 Signsoft intelliBO V3.1

Nach der zu Beginn durchgeführten Internetrecherche anhand der in Abschnitt 2.3 genannten Selektionskriterien, wurde als erste zu untersuchende JDO-Implementierung intelliBO V3.1 www.signsoft.de der Firma Signsoft aus Dresden gewählt. IntelliBO ist in Java geschrieben und stellt eine 100%ige Implementierung der JDO-Spezifikation 1.0 dar, bietet sogar über den Standard hinaus noch weitere Funktionalität an. Es ist eine kommerzielle JDO-Implementierung und unterstützt die Datenbank MySQL. Der Implementierung liegt eine graphische Entwicklungsumgebung bei, durch deren Hilfe aus vorhandenen Klassen die zugehörigen Persistenz-Deskriptoren erstellt werden können, sowie einen Tabellenskript-Generator, einen Persistenz-Deskriptor-Verifizierer[1] und einen Bytecode-Enhancer[2]. Während dem Entwurf und Implementierung der Klassen braucht sich der Entwickler keine Gedanken um die Speicherung der Objekte zu machen, sondern öffnet sein Projekt in der graphischen Entwicklungsumgebung und erstellt die zu den speichernden Klassen zugehörigen Persistenz-Deskriptoren. In diesen werden die zu speichernden Daten anhand Abbildungsbeschreibungen, d.h. wie welches Attribut des Objektes in den Tabellen der Datenbank abgelegt werden soll, beschrieben. Die Installation und Verwendung von intelliBO wird in Abschnitt 3.2.1.3 erläutert.

2.4.2 Hibernate V1.2.3

Hibernate ist ein in Java geschriebenes Objekt-Relational-Abbildungswerkzeug. Es werden alle gängigen Datenbanken, darunter MySQL und HSQL unterstützt. Es stellt keine JDO-Implementierung dar, weist aber Parallelen zur JDO-Spezifikation auf. Es ist eine Open-Source-Implementierung und unterliegt der LGPL[3]-Lizenzform, d.h. es darf in Open-Source und kommerziellen Projekten frei verwendet werden. Die aktuelle Version ist unter: http://hibernate.bluemars.net zu finden. Der Implementierung liegt keine graphische Entwicklungsumgebung bei, sondern Kommandozeilen-orientierte Hilfswerkzeuge wie z.B. Persistenz-Deskriptor- und Tabellenskript- Generatoren. Gegenüber intelliBO wird nicht der Bytecode-Enhancement-Vorgang verwendet, sondern intern nutzt Hibernate die Java Reflection-API und erzeugt dynamisch SQL-Statements, um mit den Objekten zu arbeiten. Eine Installationsanleitung ist in Abschnitt 3.2.2.4 zu finden.

[1] Mit diesem Werkzeug können die erstellten Persistenz-Deskriptoren auf Syntax-Korrektheit geprüft werden

[2] Eine Erklärung dieses Vorganges erfolgte in Abschnitt 2.2.1.2

[3] Lesser General Public License http://www.gnu.org/copyleft/lesser.html

2.4.3CASTOR JDO

Castor ist ein in Java geschriebenes Open-Source Data-Binding-Framework[4]. Neben dem XML-Data-Binding bietet es noch die Möglichkeit der objekt-relationalen Abbildung von Java-Objekten auf relationale Datenbanksysteme. Verwendet wurde die Version 0.9.4.3. Diese ist online unter: http://castor.exolab.org verfügbar. Castor JDO wurde unabhängig von der von SUN verabschiedeten JDO-Spezifikation entwickelt, weist aber einige Parallelen zu dieser auf. Der Implementierung liegen keine graphische Entwicklungsumgebung und auch keine Kommandozeilen-basierten Hilfswerkzeuge bei. Wie auch Hibernate nutzt Castor JDO intern die Java Reflection-API und erzeugt somit dynamisch SQL-Statements, um mit den Objekten zu arbeiten. Die Installation wird in Abschnitt 3.2.3.4 erklärt.

2.4.4Solarmetric Kodo JDO V2.4.3

Als weitere JDO-Implementierung wurde Kodo der Firma Solarmetric ausgewählt. Es ist eine in Java geschriebene 100%ige Implementierung der JDO-Spezifikation 1.0. Solarmetric bietet zwei kommerzielle JDO-Produkte an: Kodo JDO Enterprise Edition (3000$), diese Version wird benötigt, falls die Applikation durch einen Applikation Server verwaltet werden soll und die Kodo JDO Standard Edition (600$), ohne Unterstützung für einen Applikation Server. Für diese Version werden zusätzliche optionale Pakete, u.a. ein Performance-Add-On (dieses verspricht eine 20-40fache Performanceverbesserung [Kodo03]), ein Query-Add-On (Erweiterungen für die Abfragesprache), angeboten. Diese optionalen Pakete sind in der Enterprise Edition bereits integriert. Die Datenbanken MySQL und HSQL werden durch Kodo unterstützt. Der JDO-Implementierung liegt keine eigene graphische Entwicklungsumgebung bei, sondern Plugins für Entwicklungsumgebungen u.a. Borland JBuilder und SUN One Studio/NetBeans. Die Implementierung wird aber mit Kommandozeilen-basierten Hilfswerkzeugen, z.B. automatische Generierung der, unter Verwendung des eigenen Identitätenkonzept, benötigten Primary-Key-Klassen, einen Bytecode-Enhancer und ein Schemamanipulationstool ausgeliefert. Eine Installationsanleitung ist in Abschnitt 3.2.4.3 zu finden.

2.4.5Libelis Lido V1.4

Lido stellt eine in Java geschriebene 100%ige Implementierung der JDO-Spezifikation 1.0 dar. Die Firma Libelis bietet zwei kommerzielle Standard- und Professional- und eine zu Evaluierungszwecken Community- Edition an. Die Standard- Edition (600Euro) unterstützt nur Open-Source relationale Datenbanken und die Speicherung in Dateien. Zugang zu kommerziellen relationalen Datenbanken und die Einbettung in einen Application Server werden nur von der Professional-Edidion (2000Euro) unterstützt. Diese stellt dem Anwender auch eine graphische Entwicklungsumgebung zur Verfügung. Evaluiert wurde die Community-Edition, die die Datenbanken MySQL und HSQL unterstützt. Der Implementierung liegen Kommandozeilen-basierte Hilfswerkzeuge: einen Bytecode-Enhancer und ein Schemagenerierungstool bei. Die Installation und Verwendung von Lido wird in Abschnitt 3.2.5.3 erläutert.

[4] Mechanismus, um Daten aus einer XML-Datei in einer Java-Applikation zu verwenden

2.5 Kriterien für die Evaluierung

Im folgenden werden die zu Beginn festgelegten Evaluierungskriterien erläutert:

2.5.1 Integrationsaufwand für bereits existierende Klassen

Es soll untersucht werden, ob Änderungen an einem vorhandenen Objektmodell vorgenommen werden müssen, um die zu speichernden Daten der Applikation durch die ausgewählte JDO-Implementierung verwalten zu lassen. D.h. ob vorhandene Klassen von JDO-spezifischen Klassen erben oder ob bestimmte Schnittstellen implementiert werden müssen. Des weiteren soll untersucht werden, ob SQL-Code oder JDBC-Anweisungen in den Programmcode aufgenommen werden müssen, oder ob der Datenbankzugriff durch die JDO-Implementierung transparent für den Entwickler geschieht.

2.5.2 Performance und Ressourcenverbrauch

Es werden Zeiten und Speicherverbrauch von typischen Datenbank-Operationen[1], die durch die JDO-Implementierung durchgeführt werden, gemessen. Die Hard- und Softwarekonfiguration des verwendeten Rechners für die Messungen wird in Abschnitt 3.1 aufgelistet. Eine wichtige Erkenntnis ist hierbei, eine Relation aufstellen zu können, wie lange der Anwender durch den Automatismus[2] von JDO warten muss, bis die Datenbank-Operationen durchgeführt wurden, und ob und wie weit dadurch der Arbeitsspeicher belastet wird. Als Gegenüberstellung der insert- und query-Operationen werden die Zeiten und der Speicherverbrauch für die gleiche Anzahl (n) von Objekten durch den Vorgang der Serialisierung bzw. Deserialisierung in bzw. aus n Dateien durchgeführt, d.h. ein komplett instanziiertes Primary-Objekt wird jeweils in einer Datei abgespeichert.

2.5.3 Laufzeitschwankungen

Es soll untersucht werden, ob bei mehrmaligen Ausführen des Programms unter gleichgebliebenen Eingabeparameter die gleichen Messergebnisse hinsichtlich Zeit und Speicherverbrauch auftreten.

2.5.4 Integration objektorientierter Paradigmen

2.5.4.1 Vererbung

Es soll untersucht werden, ob und wie das Konzept der Vererbung durch die jeweilige JDO-Implementierung unterstützt wird, d.h. ob die Attribute der Klassen jeweils einer eigenen Tabelle zugeordnet werden oder ob alle beteiligten Attribute der Klassen in einer einzigen Tabelle abgespeichert werden.

2.5.4.2 Assoziationen

Es soll untersucht werden, wie die gängigen Abbildungen 1…1, 1…N und N…M durch die JDO-Implementierung gehandhabt werden, wie die Abbildung von Java-Datentypen

[1] Darunter fallen insert, delete, query und update- Operationen, jeweils erfolgreich (commit-Vorgang) und nicht-erfolgreich (rollback-Vorgang)

[2] Mit dem Begriff Automatismus ist gemeint: die automatische Abbildung der zu speichernden Daten der Objekte auf Tabellen der zugrundeliegenden Datenbank

auf Datentypen der zugrundeliegenden Datenbank erfolgt, d.h. ob ein automatisches Mapping erfolgt oder ob eine explizite Typkonvertierung vorgenommen werden muss, des weiteren ob die Nutzung von Fremdschlüsseln unterstützt wird und speziell wie die Handhabung für Implementierungen der Schnittstellen `List` und `Map` erfolgt.

2.5.4.3Komposition

Komposition sagt aus, dass zwischen Objekten eine Beziehung vorliegt, die sich als „ist Teil von“ oder „besteht aus“ beschreiben lässt. Diese Teile sind existenzabhängig vom Ganzen, so dass beim Löschen des Ganzen auch alle seine Bestandteile mit gelöscht werden müssen. In unserem Fall soll untersucht werden, ob und in wie weit dies automatisch durch die JDO-Implementierung geschieht oder ob der Anwender das kaskadierende Löschen selbst implementieren muss.

2.5.5Graphische Entwicklungsumgebung

Den meisten JDO-Implementierungen liegt eine graphische Entwicklungsumgebung oder Kommandozeilenwerkzeuge bei. Es soll deren Handhabbarkeit untersucht werden und welche zusätzlichen Funktionalitäten dem Anwender noch geboten werden, ob und wie z.B. das Mapping von existierenden Klassen auf schon existierende Tabellen unterstützt wird, ob der Implementierung ein Tabellenskript-Generator beiliegt, ob Plug-In[1] für andere hilfreiche Entwicklungstools zur Verfügung gestellt werden, wie der Enhancement-Vorgang erfolgt und ob und wie die zugrundeliegende Datenbank konfiguriert werden kann.

2.5.6Abfragesprache(n)

Es soll untersucht werden, ob über die jdo-spezifische Abfragesprache JDOQL (siehe Abschnitt 2.2.6) noch weitere wie SQL oder vom Hersteller eigene Abfragesprachen unterstützt werden.

2.5.7Unterstützte Systemumgebungen

Ob die JDO-Implementierung in einen Applikationsserver integriert werden kann oder nur in einer Client-Server-Umgebung lauffähig ist.

3. Prototyp

In diesem Abschnitt wird die jeweilige Implementierung des erstellten Prototyps anhand des Klassendiagramms und des Entity-Relationship-Modells erläutert. Des weiteren die verwendeten Datenbanken MySQL und HSQL vorgestellt und auf Besonderheiten der erstellten Persistenz-Deskriptoren eingegangen.

3.1 Systemumgebung

Die Untersuchungen wurden auf folgender Hard- und Software-Konfiguration durchgeführt:

Prozessor:	Intel Pentium III 800Mhz
Arbeitsspeicher:	512MB SD-RAM

[1] Plug-In sind Hilfsprogramme, die die Funktionalität von anderen Programmen erweitern

Festplatte:	10GB, 5400 U/min
Betriebssystem:	Windows 2000
Datenbank-Version:	MySQL 3.23.54-max-nt
JDK-Version:	SDK 1.4.1_01

3.1.1Die Datenbank MySQL

3.1.1.1Warum MySQL?

MySQL ist ein relationales Datenbankmanagementsystem. Der große Vorteil ist, dass es frei und kostenlos unter www.mysql.com erhältlich ist. MySQL ist die verbreitetste Open-Source-Datenbank, lizenziert mit GNU GENERAL PUBLIC LICENSE http://www.gnu.org/. Es ist eine echte Multi-User, Multi-Threaded SQL Datenbank und wird von mehreren großen Providern oder auch Suchmaschinenbetreibern eingesetzt. MySQL ist eine Client/Server Implementierung, die aus einem Server-Dämon mysqld-max-nt (es existieren noch andere) und vielen Client Programmen, sowie Bibliotheken für unterschiedliche Programmiersprachen besteht. Die wichtigsten Eigenschaften von MySQL sind Geschwindigkeit, Stabilität und einfache Bedienbarkeit. [MySQ03]

3.1.1.2Der Tabellentyp innoDB

Gewählt wurde der Tabellentyp innoDB, da der Standard-Tabellentyp von MySQL MyISAM keine Transaktionen unterstüzt. Es folgt eine Auflistung der Vorteile des Tabellentyp innoDB:

- es wird ein Transaktionskonzept mit commit, rollback und crash-recovery[1] Fähigkeiten unterstützt
- unterstützt als erster MySQL-Tabellentyp Fremdschlüssel
- hat seinen eigenen Buffer-Pool um Daten und Indexe im Speicher zu cachen → bessere Performance
- InnoDB erkennt automatisch Deadlocks[2] während einer Transaktion und führt automatisch ein rollback aus
- ist gegenüber MyISAM sicherer, d.h. wenn MySQL zusammenbricht oder es zu Hardware-Problemen kommt, erhält man die Daten durch automatisches Recovery oder von einem Backup + zugehörige Transaktionslog-Dateien zurück

3.1.2Die Datenbank Hypersonic SQL (HSQL)

In die Evaluierung wurde noch eine weitere Datenbank names HSQL eingezogen. Hypersonic SQL ist eine in Java geschriebene relationale Open-Source Datenbank und kann unter http://hsqldb.sourceforge.net/ bezogen werden. Unterstützt werden „in memory-“ und plattenbasierte Tabellen. Die Datenbank kann als Server oder aber

[1] Die Fähigkeit, dass die Datenbank nach einem Systemabsturz die Arbeit ohne Datenverlust wiederaufnimmt

[2] Dt.Verklemmung: wenn z.B. eine Sperre auf zwei Objekten vorliegt, und gegenseitig versucht wird auf das dann gesperrte Objekt zu zugreifen.

eingebettet in der gleichen Java Virtual Machine, wie die sie verwendende Java-Applikation, ausgeführt werden. Fremdschlüssel und Transaktionen werden unterstützt. Eine detaillierte Installations- bzw. Konfigurationsanleitung ist in Abschnitt 6.5 zu finden.

3.2 Klassendiagramm und Implementierung

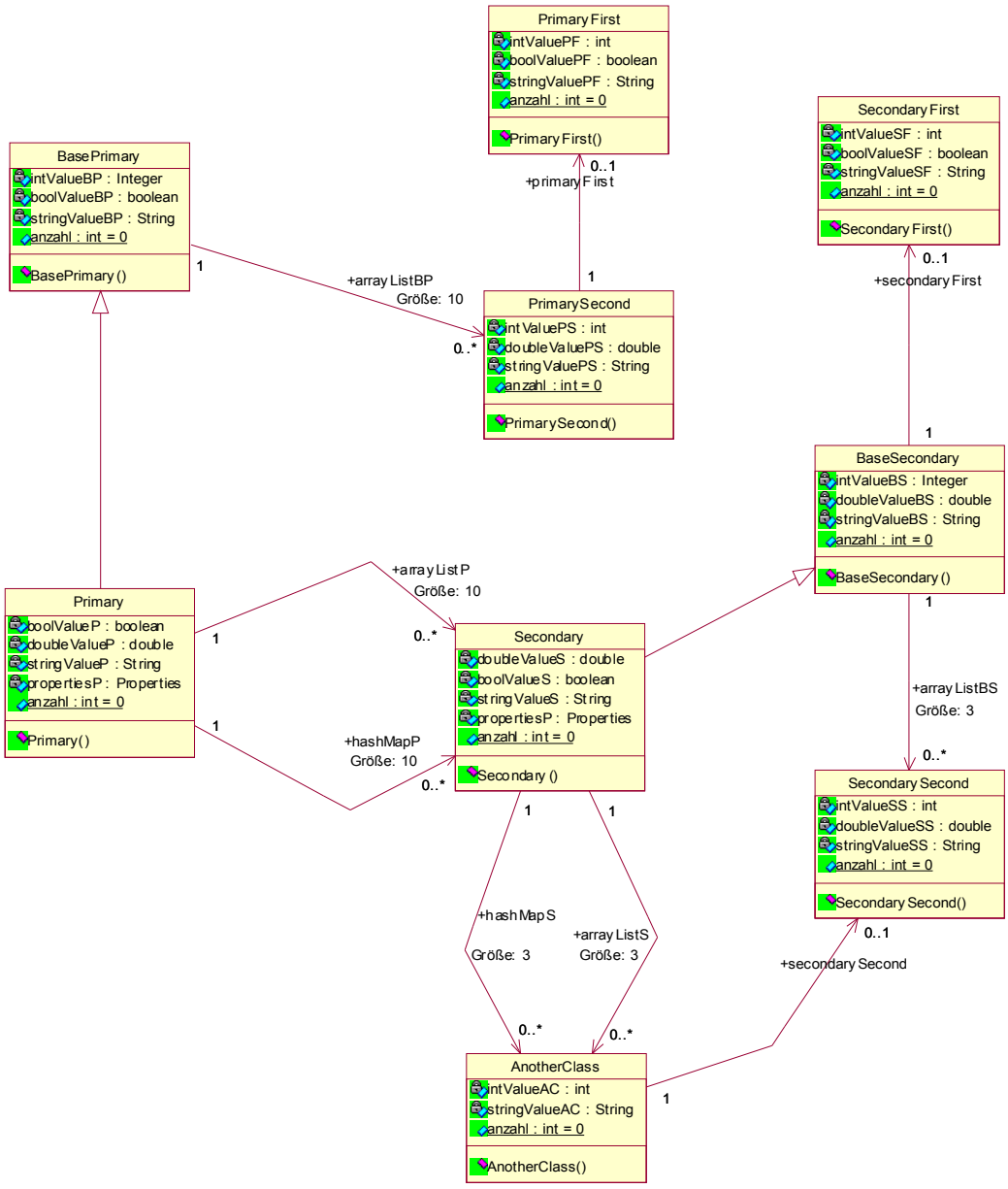


Abbildung 6„Klassendiagramm des Prototyp“

Obiges Klassendiagramm stellt die Grundlage für die Evaluierung von JDO dar. Es wird untersucht ob und wie die verschiedenen Datentypen, Vererbung und Assoziationen von den jeweiligen JDO-Implementierungen unterstützt bzw. abgebildet werden. Die Klasse `Primary` stellt das Hauptobjekt dar, welches persistent abgelegt werden soll. Es wurden absichtlich einfache Namen für die Klassen und die jeweiligen Attribute gewählt. Zu jeder Klasse des Modells wurde noch eine zugehörige PrimaryKey-Klasse

(siehe Abschnitt 2.2.1.4) erstellt, weil die Identität der Objekte von der Applikation verwaltet werden soll. Dazu wurde in jeder Klasse ein Klassenattribut `anzahl` definiert, was bei jeder Objekterzeugung im Konstruktoraufruf inkrementiert wird und dem jeweiligen int-Attribut `intValueXX`[1] zugewiesen wird. In den Klassen wurden primitive Attribute, Strings, Properties, ArrayListen und HashMaps verwendet. Die Datentypen `ArrayList` und `HashMap` stellen 1…N-Assoziationen dar, d.h. ein Objekt einer Klasse verwaltet bis zu n Objekte einer anderen Klasse. Diese Beziehung besteht als `ArrayList` der Dimension 10 zwischen Objekten der Klassen `BasePrimary` und `PrimarySecond` und zwischen Objekten der Klassen `Primary` und `Secondary`. Des weiteren besitzt die Klasse `Primary` noch eine `HashMap`, in der 10 `Secondary`-Objekte verwaltet werden. Zu Beginn der Performance- bzw. Speichertests wurde als Dimension für alle ArrayListen und HashMaps 10 gewählt und die Klassen `Secondary` und `SecondarySecond` besaßen noch `Properties` der Größe 10. 1000 `Primary`-Objekte ließen den Speicher auf über 300MB ansteigen und es kam zu einem `java.lang.OutOfMemoryError`[2], was auf die hohe Anzahl von `String`-Objekten[3] zurück zuführen ist. Aus diesem Grunde wurde die Dimension der `hashMapS`, `arrayListS` und `arrayListBS` auf 3 reduziert und nur noch `Properties` der Größe 10 in der Klasse `Primary` verwendet.

1 Primary-Objekt besteht nun aus folgenden Objekten:

- 1 BasePrimary
- 10 PrimarySecond
- 10 PrimaryFirst
- 20 Secondary
- 20 BaseSecondary
- 20 SecondaryFirst
- 120 AnotherClass
- 180 SecondarySecond

Werden jetzt 1000 `Primary`-Objekte erzeugt und in einer `ArrayList` zwischengespeichert, beansprucht diese Datenhaltung ca. 60MB im Speicher. Der Grund wieso diese n `Primary`-Objekte zu Beginn in einer `ArrayList` gespeichert werden ist, dass zu Programmstart alle Objekte instanziiert sind und dann innerhalb einer Transaktion durch die JDO-Implementierung oder dem Objekt-Relationalen-Abbildungswerkzeug persistent abgelegt werden sollen.

Es wurde ein Testfall erstellt, der zu Beginn n `Primary`-Objekte initialisiert, in einer `ArrayList` verwaltet und innerhalb einer Transaktion in die verwendetete Datenbank MySQL bzw. HSQL abspeichert. An diesen sich im Speicher befindlichen `Primary`-Objekte wurden Veränderungen innerhalb einer Transaktion durchgeführt, alle `Primary`-Objekte in der Datenbank gelöscht und `Query`-Abfragen an die Datenbank abgesetzt. Alle Tests wurden 5-10 mal mit den gleichen Eingabeparametern ausgeführt.

[1] XX steht als Abkürzung für die zugehörige Klasse, z.B. PS für `PrimarySecond`

[2] Um einem `java.lang.OutOfMemory`- Error vorzubeugen, startet man den Java-Interpreter java.exe mit der `-Xmx` Option, um den der Java Virtual Machine zur Verfügung stehenden Heap-Speicher zu erhöhen

[3] Der Datentyp `String` wird unter Java im Speicher als ein Objekt bestehend aus Header, Offset usw. abgelegt, und benötigt somit viel Arbeitsplatz

Es wurden keine drastischen Schwankungen festgestellt. In Abschnitt 4 ist der jeweils resultierende Mittelwert der durchgeführten Operationen zu finden.

3.2.1Implementierung intelliBO

Für die Evaluierung von intelliBO wurde obiges Objektmodell angepasst. Die Oberklassen `BasePrimary` und `BaseSecondary` verwenden nun `Integer` als Datentyp für die ObjektID (`intValueXX`). Der Grund dafür ist, dass unter intelliBO bei Verwendung des Datentyp `HashMap` kein eigener Schlüssel angegeben werden kann, sondern dass immer automatisch die ObjektID des in dieser `Map` abgelegten Objektes als Schlüssel genommen wird. Dieses Problem soll in den nächsten Versionen von intelliBO behoben worden sein.

Des weiteren wurde eine Hilfsklasse `ConfJDO` erstellt, die statische Methoden beinhaltet. Durch diese wird die Schnittstelle `PersistenzManagerFactory` konfiguriert und der Anwendung ein `PersistenzManager` zur Verfügung gestellt. Die Konfiguration bindet somit alle erzeugten `PersistenzManager` an die zugrunde liegende Datenbank, d.h. es wird der Standard-JDBC[1] und der erweiterte[2] JDBC-Datenbanktreiber von Signsoft eingebunden, die URL, Benutzername und Password der Datenbank gesetzt und festgelegt welches Transaktionskonzept (pessimistisch oder optimistisch) verwendet werden soll. Diese Angaben sind als Konstanten in der Klasse `ConfJDO` definiert.

Die Testklasse `Test` bezieht von der Hilfsklasse `ConfJDO` einen `PersistenceManager` und führt die untersuchten Tests durch. Es wurden Methoden für die Serialisierung, Deserialisierung, insert-, delete-, query- und update- Operationen implementiert (eine Erklärung dieser Methoden ist in der beiliegenden JavaDoc zu finden).

Vor Ausführung der Tests muss noch eine Datenbank diplom in MySQL bzw. HSQL erstellt und die entsprechenden Tabellen in dieser angelegt werden. (siehe Abschnitte 6.4 und 6.5). Die Beziehungen der angelegten Tabellen werden im nächsten Abschnitt anhand eines Entity-Relationship-Modells dargestellt.

[1] Es wurden Tests mit dem JDBC-Treiber von Mark Matthews in der Version 2.0.4 [MaMa02] und Tests mit dem offiziellen JDBC-Treiber von MySQL Connector/J V3.0 [Conn03] durchgeführt

[2] IntelliBO unterstützt eine Vielzahl von Datenbanken mit speziellen Treibern, die zusätzliche Funktionen zur Verfügung stellen und bekannte Probleme mit den Originaltreibern umgehen.

3.2.1.1 Entity-Relationship-Modell

Es folgt ein Entity-Relationship-Modell der Tabellenbeziehungen der Datenbank diplom, auf denen die Evaluierung basierte. Die **Primärschlüssel** sind fett-unterstrichen und die **Fremdschlüssel** fett dargestellt.

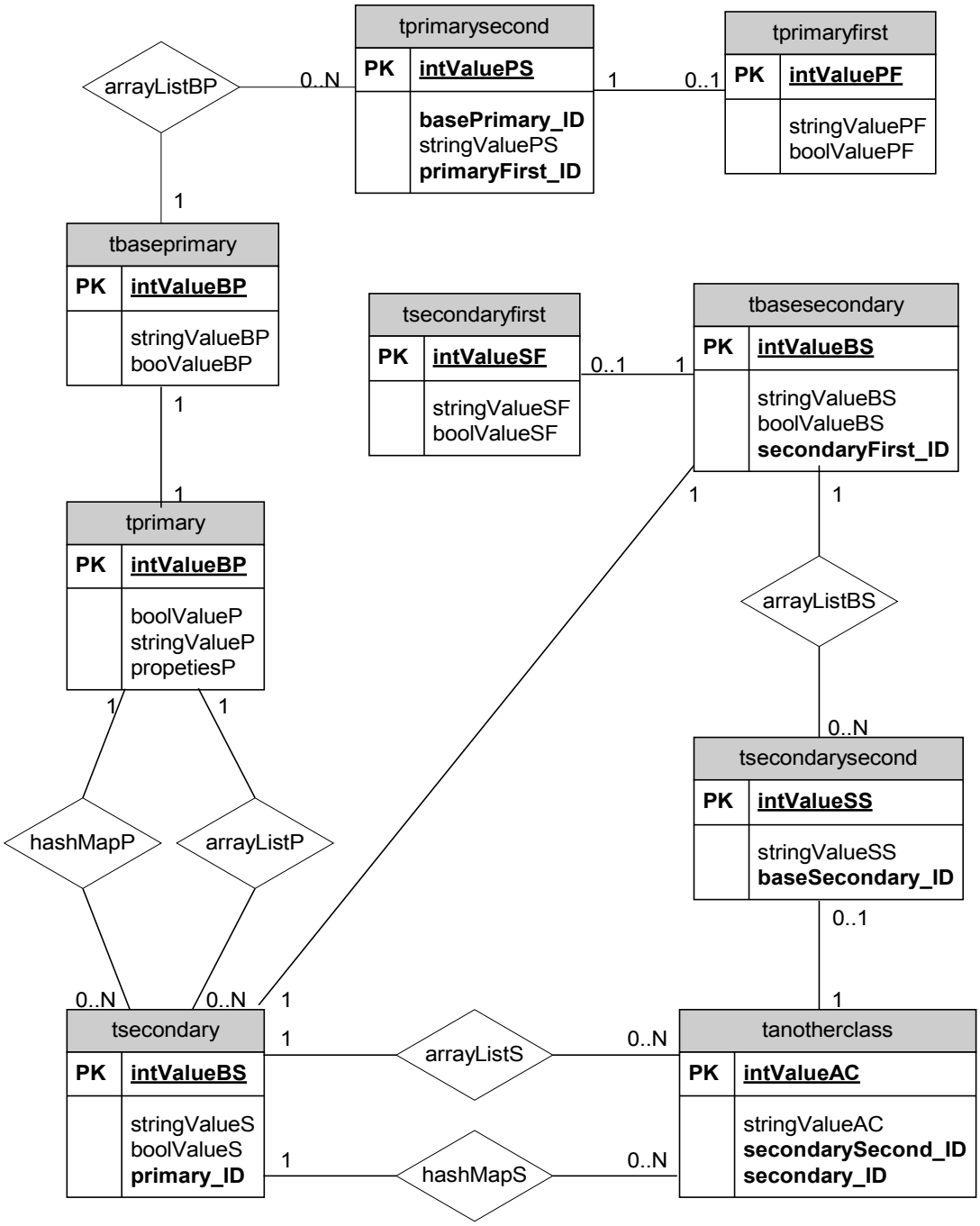


Abbildung 7 „Entity-Relationship-Modell der Datenbank diplom“

Der Grund, dass eine Doppelrelation zwischen den Tabellen `tprimary` und `tsecondary` und zwischen `tsecondary` und `tanotherclass` erstellt wurde, wird in Abschnitt 4.1.5 erläutert.

3.2.1.2Persistenz-Deskriptor am Beispiel Primary.jdo

Zu jeder Klasse unseres Objektmodells wurde ein Persistenz-Deskriptor erstellt, der der JDO-Implementierung mitteilt wo und wie die Objekte in der Datenbank abgelegt werden sollen. In diesem Abschnitt wird der zur Klasse `Primary` gehörende Persistenz-Deskriptor für intelliBO V3.2 erläutert. Alle erstellten Persistenz-Deskriptoren sind auf der Projekt-CD zu finden.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="prototype.model">
    <class identity-type="application" name="Primary" persistence-capable-superclass="BasePrimary"
        requires-extent="true">
     <field name="stringValueP">
        <extension key="jdbc" vendor-name="ssibo">
        <extension key="mapping" value="direct" vendor-name="ssibo"/>
        </extension>
     </field>
     <field name="boolValueP">
        <extension key="jdbc" vendor-name="ssibo">
        <extension key="mapping" value="direct" vendor-name="ssibo"/>
        </extension>
     </field>
     <field name="hashMapP">
        <map key-type="java.lang.Integer" value-type="prototype.model.Secondary"/>
        <extension key="jdbc" vendor-name="ssibo">
        <extension key="mapping" value="one-to-many" vendor-name="ssibo">
        <extension key="result-type" value="java.util.HashMap" vendor-name="ssibo"/>
        <extension key="element-type" value="prototype.model.Secondary" vendor-name="ssibo"/>
        <extension key="cascade-delete" value="true" vendor-name="ssibo"/>
        </extension>
        </extension>
     </field>
     <field name="arrayListP">
        <collection element-type="prototype.model.Secondary"/>
        <extension key="jdbc" vendor-name="ssibo">
        <extension key="mapping" value="one-to-many" vendor-name="ssibo">
        <extension key="result-type" value="java.util.ArrayList" vendor-name="ssibo"/>
        <extension key="element-type" value="prototype.model.Secondary" vendor-name="ssibo"/>
        <extension key="cascade-delete" value="true" vendor-name="ssibo"/>
        <extension key="reference" vendor-name="ssibo">
        <extension key="src-table" value="TPRIMARY" vendor-name="ssibo"/>
        <extension key="src-field" value="FINTVALUEBP" vendor-name="ssibo"/>
        <extension key="dst-table" value="TSECONDARY" vendor-name="ssibo"/>
        <extension key="dst-field" value="FTPRIMARY_ARRAY" vendor-name="ssibo"/>
        </extension>
        </extension>
        </extension>
     </field>
     <field name="propertiesP" persistence-modifier="persistent">
       <extension key="jdbc" vendor-name="ssibo">
         <extension key="mapping" value="serial" vendor-name="ssibo">
          <extension key="table-name" value="TPRIMARY" vendor-name="ssibo"/>
          <extension key="field-name" value="FPROPERTIESP" vendor-name="ssibo"/>
          <extension key="sql-type" value="BLOB" vendor-name="ssibo"/>
         </extension>
       </extension>
     </field>
     <extension key="jdbc" vendor-name="ssibo"/>
    </class>
  </package>
</jdo>
```

Abbildung 8„Persistenz-Deskriptor der Klasse Primary“

Es folgt eine Erläuterung der einzelnen Tags:

```
<package name="prototype.model">
```

Durch das Attribut `name` des Elementes `<package>` wird der Name des Package[1], in welchem sich die Klasse befindet eingeführt.

```
<class identity-type="application" name="Primary" persistence-capable-superclass="BasePrimary" requires-extent="true">
```

Das Element `<class>` wird verwendet um eine persistente Klasse zu beschreiben, d.h.: diese Klasse implementiert die Schnittstelle `PersistenceCapable`. Das Attribut `identity-type` legt den Typ der Objektidentität fest. In diesem Fall das eigene Identitätenkonzept. Durch das Attribut `persistence-capable-superclass` wird angegeben, dass diese Klasse von der Klasse `BasePrimary` erbt. D.h. es muss nicht explizit eine PrimaryKey-Klasse für diese Klasse erstellt werden, da diese von der Oberklasse geerbt wird. Die Klasse `Primary` erbt den PrimaryKey (`intValueBP`) der Oberklasse. Durch setzen von `requires-extent` auf `true`, ist es möglich über einen `Extent`[2] auf diese Klasse zuzugreifen.

```
<field name="stringValueP">
  <extension key="jdbc" vendor-name="ssibo">
  <extension key="mapping" value="direct" vendor-name="ssibo"/>
  </extension>
</field>
```

Durch das Element `<field>` werden die persistenten Attribute einer Klasse beschrieben. Die JDO-Spezifikation definiert Regeln um persistene Felder einer Klasse automatisch zu erkennen und Standardwerte für deren Attribute zu setzen, falls diese nicht explizit im Persistenz-Deskriptor erwähnt werden. [Russ02, S. 125]
Wie bereits im Abschnitt 2.2.3 erwähnt, werden durch das Element `<extension>` herstellerspezifische Informationen für den Enhancement-Vorgang eingeführt. Diese werden durch das Attribut `vendor-name` eindeutig einem JDO-Anbieter zugeordnet, so dass der erstellte Persistenz-Deskriptor auch von anderen JDO-Implementierungen genutzt werden kann. Im obigen Beispiel bedeutet `ssibo` die JDO-Implementierung der Firma Signsoft names intelliBO. Signsoft definiert das Element `<extension>` als Schlüssel-Wert-Paar. Durch den Schlüssel `mapping` werden die Abbildungsregeln der einzelnen Attribute auf Spalten in den Tabellen definiert. Der Wert `direct` sagt aus, dass das Attribut `stringValueP` der Klasse `primary` direkt auf eine Spalte in der Tabelle abgebildet wird.

```
<field name="hashMapP">
    <map key-type="java.lang.Integer" value-type="prototype.model.Secondary"/>
    <extension key="jdbc" vendor-name="ssibo">
    <extension key="mapping" value="one-to-many" vendor-name="ssibo">
    <extension key="result-type" value="java.util.HashMap" vendor-name="ssibo"/>
```

[1] Bei größeren Projekten werden die Klassen der Übersichtlichkeit in Pakete aufgeteilt

[2] Die Klasse Extent wurde im Abschnitt 2.2.2 eingeführt

```
    <extension key="element-type" value="prototype.model.Secondary" vendor-name="ssibo"/>
    <extension key="cascade-delete" value="true" vendor-name="ssibo"/>
    </extension>
    </extension>
</field>
```

Das Attribut `hashMapP` der Klasse `Primary` ist vom Datentyp `HashMap`. Dieser implementiert die Schnittstelle `Map` und wird durch das Element `<map>` beschrieben. Eine `Map` beschreibt eine Wert-Schlüssel-Beziehung und stellt eine 1...N-Beziehung (one-to-many) dar. Durch das Attribut `key-type` des Elementes `<map>` wird der Datentyp der Schlüssel der `Map` angegeben und durch `value-type` der Datentyp der zugehörigen Werte. Zur Zeit kann bei intelliBO als Datentyp für den Schlüssel der `Map` nur der gleiche Datentyp wie auch der für die ObjektID angegeben werden. Dieses Problem wird im Abschnitt 4.1.5 näher erläutert. Der Wert des Schlüssels `result-type` definiert die verwendete Implementierung der Schnittstelle `Map` und der Wert des Schlüssel `element-type` definiert den Datentyp, der in dieser `Map` abgelegten Objekte. Durch Setzen des Schlüssels `cascade-delete` auf `true` wird erreicht, dass wenn das `Primary`-Objekt gelöscht wird, auch die Elemente in der `HashMap` mitgelöscht werden.

```
<field name="arrayListP">
    <collection element-type="prototype.model.Secondary"/>
    <extension key="jdbc" vendor-name="ssibo">
    <extension key="mapping" value="one-to-many" vendor-name="ssibo">
    <extension key="result-type" value="java.util.ArrayList" vendor-name="ssibo"/>
    <extension key="element-type" value="prototype.model.Secondary" vendor-name="ssibo"/>
    <extension key="cascade-delete" value="true" vendor-name="ssibo"/>
    <extension key="reference" vendor-name="ssibo">
    <extension key="src-table" value="TPRIMARY" vendor-name="ssibo"/>
    <extension key="src-field" value="FINTVALUEBP" vendor-name="ssibo"/>
    <extension key="dst-table" value="TSECONDARY" vendor-name="ssibo"/>
    <extension key="dst-field" value="FTPRIMARY_ARRAY" vendor-name="ssibo"/>
    </extension>
    </extension>
    </extension>
</field>
```

Das Attribut `arrayListP` der Klasse `Primary` ist vom Datentyp `ArrayList` und implementiert die Schnittstelle `Collection`. Durch Angabe des Elementes `<collection>` wird der Datentyp der in dieser `ArrayList` verwalteten Elemente angegeben. Die `ArrayList` stellt eine 1...N-Beziehung dar. In diesem Fall wird das Mapping explizit durch das Element `<reference>` angegeben, d.h.: es wird ein Fremdschlüssel `FTPRIMARY_ARRAY` in der Tabelle `TSECONDARY` definiert, der in Relation mit dem Schlüssel `FINTVALUEBP` der Tabelle `TPRIMARY` steht. Hierbei trat ein Problem auf, welches im Abschnitt 4.1.5 erläutert wird.

```
<field name="propertiesP" persistence-modifier="persistent">
    <extension key="jdbc" vendor-name="ssibo">
    <extension key="mapping" value="serial" vendor-name="ssibo">
    <extension key="table-name" value="TPRIMARY" vendor-name="ssibo"/>
    <extension key="field-name" value="FPROPERTIESP" vendor-name="ssibo"/>
    <extension key="sql-type" value="BLOB" vendor-name="ssibo"/>
    </extension>
    </extension>
</field>
```

Das Attribut `propertiesP` der Klasse `Primary` ist vom Datentyp `Properties`. Es wird mit Hilfe des Java-Serialisierungsmechanismus in der Datenbank in der Tabelle

`TPRIMARY` in der Spalte `FPROPERTIESP` vom SQL-Datentyp `BLOB` abgespeichert. Der Nachteil dieser Vorgehensweise wird im Abschnitt 4.1.5 erläutert.

3.2.1.3Installation + Benutzeranleitung

3.2.1.3.1 Systemvoraussetzungen Software

1. MySQL muss installiert, eine Datenbank `diplom`, die zugehörigen Tabellen angelegt worden sein (Anleitung siehe Abschnitt 6.4), und der verwendete JDBC-Treiber in den CLASSPATH aufgenommen werden

2. Signsoft intelliBO muss installiert, eine gültige Lizenz für die Verwendung von intelliBO vorliegen und die zugehörigen JAR-Files in den CLASSPATH aufgenommen werden

Der CLASSPATH muss folgende JAR-Files beinhalten:

%intelliHOME% = c:\signsoft\intelliBO3\redist\

%intelliHOME% dbsupport_mysql.jar;	[angepasster Datenbanktreiber]
%intelliHOME% ibocore.jar;	[Signsoft intelliBO Kern]
%intelliHOME% ibometa.jar;	[Metadata-Bibliothek]
%intelliHOME% ssjava.jar;	[Signsoft Java-Grundbibliothek]
%intelliHOME% connector.jar;	[Connector-Architektur]
%intelliHOME% ssjaxb.jar;	[JAXB-Runtime XML Data Binding]
%intelliHOME% jdbc2_0stext.jar;	[JDBC 2.0 Erweiterungen]
%intelliHOME% jdo.jar;	[JDO Spezifikation]
%intelliHOME% jta.jar;	[Transaktionssteuerung]
c:\prototype\model\Lizenz;	[intelliBO-Lizenzdatei]

je nachdem, welchen JDBC-Treiber man verwenden möchte:

c:\prototype\JDBC\mm.mysql-2.0.4-bin.jar;	[MM-SQL-Treiber V2.0.4]
c:\prototype\JDBC\mysql-connector-java-3.0.jar;	[Connector/J V3.0]

3. Die Systemvariable PATH muss jeweils das bin-Verzeichnis des installierten JDK und von Signsoft intelliBO beinhalten. Dies ist wichtig, damit die der Projekt-CD beiliegende Batch-Datei den Java-Compiler und -Interpreter und den Enhancer von intelliBO ausführen kann [siehe Anhang Abschnit xcx]

3.2.1.3.2 Systemvoraussetzungen Hardware

Die Tests wurden auf den im Abschnitt 3.1 genannten Rechner ausgeführt. Da die JDO-Implementierung den Arbeitsspeicher stark belastet[1], soll mindestens 512MB Arbeitsspeicher zur Verfügung stehen.

3.2.2Implementierung Hibernate

Im folgendem werden die wichtigsten Klassen und Schnittstellen der Hibernate-API kurz vorgestellt:

[1] Auf dieses Problem wird im Abschnitt 4.2 eingegangen

3.2.2.1Wichtige Klassen und Schnittstellen der Hibernate-API

`Datastore:`

Durch eine Instanz der Schnittstelle `Datastore` wird der erstellte Persistenz-Deskriptor, d.h. die Abbildungsregeln der Java-Klassen auf die zugrundeliegende relationale Datenbank, der Applikation zugänglich gemacht. Durch die Methode `storeFile(String xmlFile)` wird der übergebene XML-Dateiname im Classpath gesucht und die entsprechende XML-Datei eingelesen und ausgewertet.

`SessionFactory:`

Eine Instanz der Schnittstellte `SessionFactory` enthält alle Konfigurationen der verwendeten Datenbank, die über die Methode `buildSessionFactory()` der Schnittstelle `Datastore` aus der sich im Classpath befindlichen Datei `hibernate.properties`[1] ausgelesen werden. Über die Methode `openSession()` der Schnittstelle `SessionFactory` wird eine `Session` erzeugt. Falls mehrere Datenbanken verwendet werden, wird pro Datenbank eine `SessionFactory` instanziiert. Instanzen der Schnittstelle `SessionFactory` sind „threadsafe[2]“ und die an diese gebundenen Datenbankkonfigurationen können nicht mehr geändert werden.

`Session`:

Die Schnittstelle `Session` kapselt die JDBC-Verbindung zu der zugrundeliegenden Datenbank und dient als Quelle für Transaktionen. Die Lebensdauer einer `Session`-Instanz ist an den Beginn und das Ende einer lokalen Transaktion gebunden. Es werden Methoden zum Suchen, Löschen, Eintragen und Finden bereitgestellt.

`Transaction`:

Die Schnittstelle `Transaction` dient dazu, atomare Einheiten von Operationen zu bilden. Eine Transaktion `trans` wird durch Aufruf der Methode `beginTransaction()` der Schnittstelle `Session` eingeleitet. Durch `trans.commit()` wird die `Session` abgeschlossen und die Transaktion beendet. Ein `rollback()` bricht die aktuelle Transaktion ab und verwirft die in der aktuellen Transaktion durchgeführten Änderungen.

`Query:`

Eine Instanz der Schnittstelle `Query` stellt eine objekt-orientierte Abfragesprache dar. Diese wird zur Laufzeit in SQL-Code umgewandelt. Durch Aufruf der Methode `createQuery(String queryString)` wird eine `Query`-Instanz erzeugt.

`Hibernate:`

Die Klasse `Hibernate` stellt statische Methoden zum Zugriff auf die Hibernate-Datentypen für die `Query`-Abfragen bereit, erzeugt eine neue nicht-initialisierte

[1] Die Bestandteile der Datei hibernate.properties werden weiter unten erläutert

[2] Wird sicher gestellt, dass wenn ein Thread auf die Daten des Objektes zugreift, diese nicht durch einen anderen Thread der auch auf diese Daten zugreift, geändert werden können.

`Datastore`-Instanz und bietet die Möglichkeit während der Laufzeit der Applikation ein „lazy[1]“ initialisiertes Objekt vollständig zu initialisieren.

`Lifecycle`:

Um ein Callback-Mechanismus[2] für die Lösch-, Eintrage-, Speicher- und Update-Operationen bereitzustellen, muss die Schnittstelle `Lifecycle` implementiert werden.

Gegenüber der Implementierung von intelliBO wurden für die ObjektIDs aller Klassen der primitive Datentyp `int` verwendet. Die Klasse `HibernateTest` beinhaltet alle wichtigen Methoden für die Evaluierung, die in der beiliegenden Java-Doc erläutert werden.

Vor Ausführung der Tests muss noch eine Datenbank `hibernate` in MySQL bzw. HSQL erstellt und die entsprechenden Tabellen in dieser angelegt werden (siehe Abschnitt 6.4 bzw 6.5). Im nächsten Abschnitt werden die Beziehungen der angelegten Tabellen anhand eines Entity-Relationship-Modells erläutert.

Die Datei „`hibernate.properties`“, die die Datenbankkonfiguration beinhaltet:

Ersetzungsregeln, wie bestimmte Werte in der Datenbank abgebildet werden
hibernate.query.substitutions true 1, false 0, yes 'Y', no 'N'

Import-Anweisung für Package`s, d.h innerhalb einer Query muss nicht mehr der komplette Package-
Name angegeben werden
hibernate.query.imports hibernate

Hibernate wird die verwendete Datenbank bekannt gegeben
hibernate.dialect cirrus.hibernate.sql.MySQLDialect

der verwendete JDBC-Treiber wird eingeführt
hibernate.connection.driver_class com.mysql.jdbc.Driver

URL der Datenbank
hibernate.connection.url jdbc:mysql://localhost/hibernate

Benutzername für die Datenbank
hibernate.connection.username timo

Password der Datenbank
hibernate.connection.password timo

Aktiviert die Anzeige der dynamisch erzeugten SQL-Statements
hibernate.show_sql true

gibt die Transaktionsart an
hibernate.transaction.factory_class cirrus.hibernate.transaction.JDBCTransactionFactory

[1] Der Begriff „Lazy Initialization“ sagt aus, dass die von diesem Objekt aus referenzierten Objekte noch nicht initialisiert sind.

[2] Der Begriff Callback-Mechanismus wurde in Abschnitt 2.2.4.5 eingeführt.

3.2.2.2Entity-Relationship-Modell

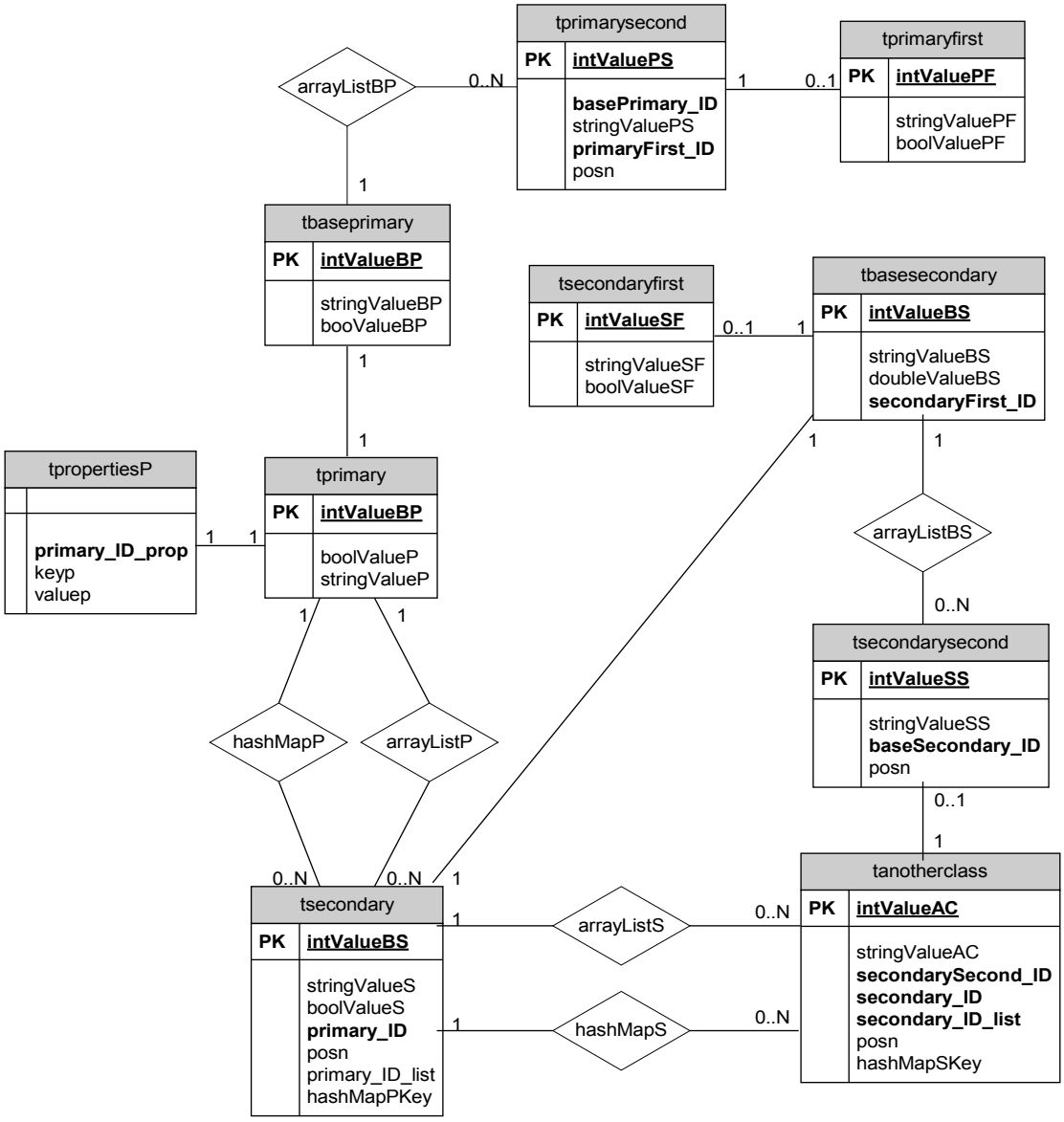


Abbildung 9„Entity-Relationship-Modell der Datenbank hibernate“

Im obigen Modell sind die **Primärschlüssel** fett-unterstrichen und die **Fremdschlüssel** fett dargestellt.

Es ist zu erkennen, dass die `Properties` der Klasse `Primary` in einer eigenen Tabelle abgelegt werden. Dies hat gegenüber der Verwendung eines `BLOBs` den Vorteil, dass nun gezielt nach einem Schlüssel gesucht werden kann.

Des weiteren wird der zur `HashMap` gehörende Schlüssel auch in den Tabellen der entsprechenden Klassen aufgelistet.

Die Abbildungsregeln von Hibernate erzwingen, dass die Tabelle der Klasse, die eine `List` als Attribut beinhaltet, eine zusätzliche Spalte zur Positionsbestimmung des Elementes der Liste führt.

3.2.2.3Persistenz-Deskriptor

Im Anhang 6.6 ist der erstellte Persistenz-Deskriptor „`prototype.hbm.xml`“ zu finden. Es folgt eine Erläuterung ausgewählter Elemente:

```
<class name="hibernate.AnotherClass" table="tanotherclass">
    <id name="intValueAC" type="int" column="intValueAC">
        <generator class="assigned"/>
    </id>
    <many-to-one name="secondarySecond" column="secondarySecond_ID"
        class="hibernate.SecondarySecond" cascade="delete" />
    <property name="stringValueAC" column="stringValueAC" type="string"/>
</class>
```

Das Element `<class>` beschreibt die zu speichernde Klasse und weist dieser durch das Attribut `table` die zugehörige Tabelle zu. Durch das Element `<id>` wird die ObjektID der Objekte der entsprechenden Tabelle festgelegt. Es besteht die Möglichkeit durch Angabe des Attributes `generator class` anzugeben, wie die Identität-Verwaltung der zugehörigen Objekte erfolgt. `assigned` sagt aus, dass die Applikation für die Vergabe der ObjektIDs verantwortlich ist. Das Element `<property>` definiert die Attribute, die direkt auf Spalten der Tabelle abgebildet werden. Durch Angabe des Elementes `<many-to-one>` wird eine 1...1-Assoziation eingeleitet. Das Attribut `name` deklariert den Namen der Objektreferenz, auf die die Assoziation gerichtet ist. Durch Angabe von `column` wird die Spalte, die den Fremdschlüssel beinhaltet beschrieben. `class` definiert den Typ der Objektreferenz. Durch Angabe von `cascade="delete"` wird erreicht, dass wenn das Hauptobjekt gelöscht wird, auch die von diesem aus referenzierten Objekte mit gelöscht werden.

```
<list role="arrayListBP" table="tprimarysecond" lazy="false" cascade="delete">
    <key column="basePrimary_ID"/>
    <index column="posn" type="int"/>
    <one-to-many class="hibernate.PrimarySecond"/>
</list>
```

Durch das Element `<list>` wird festgelegt wie Datentypen, die die Schnittstelle `List` implementieren, abgebildet werden. Dabei definiert das Attribut `role` den Variablennamen, `table` den Tabellenname der Tabelle, die den Fremdschlüssel (eingeführt durch das Attribut `key`) und die Elementposition (eingeführt durch das Attribut `index`) beinhaltet. Durch Angabe des Attributes `one-to-many` wird die Klasse der Elemente der `List` bekannt gemacht. Durch Angabe von `lazy=true` oder `false` wird angegeben, ob „Lazy Initialization“ verwendet werden soll. `True` bedeutet, dass die referenzierten Objekte erst bei Bedarf aus der Datenbank geladen werden, wo hingegen durch Angabe von `false` der komplette Objektgraph[1] initialisiert wird.

```
<class name="hibernate.BasePrimary" table="tbaseprimary">
……..
    <joined-subclass name="hibernate.Primary" table="tprimary">
                <key column="intValueBP"/>
……..
        <map role="hashMapS" table="tanotherclass" lazy="false" cascade="delete">
            <key column="secondary_ID"/>
            <index column="hashMapSKey" type="int"/>
            <one-to-many class="hibernate.AnotherClass"/>
        </map>
    </joined-subclass>
```

[1] Ausgehend vom Hauptobjekt werden alle von diesem aus referenzierten Objekte initialisiert

```
</class>
```

Durch Angabe des Elementes `<joined-subclass>` innerhalb des Elementes `<class>` wird die Unterklasse der zugehörigen Oberklasse, beschrieben durch das äußere `<class>` bekannt gegeben. `<joined-subclass>` sagt aus, dass die Unterklasse in einer eigenen Tabelle `table` abgebildet wird und `key` gibt den Fremdschlüssel auf den Primärschlüssel der Oberklasse an. Das Element `<map>` beschreibt wie Datentypen, die die Schnittstelle `Map` implementieren, abgebildet werden. Hierbei werden in der Spalte, eingeführt durch das Attribut `index`, die Schlüssel der `Map` gespeichert.

3.2.2.4Installation + Benutzeranleitung

3.2.2.4.1 Systemvoraussetzungen Software

1. MySQL bzw. HSQL muss installiert, eine Datenbank `hibernate`, die zugehörigen Tabellen angelegt worden sein (Anleitung siehe Abschnitt 6.4 bzw. 6.5), und der verwendete JDBC-Treiber in den CLASSPATH aufgenommen werden.

2. Hibernate liegt als eine Sammlung von JAR-Files vor, die alle in den CLASSPATH aufgenommen werden müssen:

Der CLASSPATH muss folgende JAR-Files beinhalten:

%hibernateHOME% = c:\hibernate-1.2\lib\

%hibernateHOME % c3p0.jar;
%hibernateHOME % cglib.jar;
%hibernateHOME % commons-collections.jar;
%hibernateHOME % commons-dbcp.jar;
%hibernateHOME % commons-lang.jar;
%hibernateHOME % commons-logging.jar;
%hibernateHOME % commons-pool.jar;
%hibernateHOME % j2ee.jar;
%hibernateHOME % jdom.jar;
%hibernateHOME % junit.jar;
%hibernateHOME % odmg.jar;
%hibernateHOME % xalan.jar;
%hibernateHOME % xerces.jar;
%hibernateHOME % xml-apis.jar;
%hibernateHOME % hibernate.jar;

c:\hibernate\JDBC\mysql-connector-java-3.0.jar; [Connector/J V3.0]

3. Auch die Konfigurationsdateien „`hibernate.properties`“ und „`prototype.hbm.xml`“ müssen in den CLASSPATH aufgenommen werden.

3.2.2.4.2 Systemvoraussetzungen Hardware

Die Tests wurden auf dem im Abschnitt 3.1 genannten Rechner ausgeführt.

3.2.3Implementierung Castor

Im folgendem werden die wichtigsten Klassen und Schnittstellen der Castor-API kurz vorgestellt:

3.2.3.1Wichtige Klassen und Schnittstellen der Castor-API

JDO:

Durch die Methode `setDatabaseName(String name)` wird der Applikation die verwendete relationale Datenbank bekanntgegeben sowie die entsprechenden Datenbankkonfigurationsdaten, die durch die Methode `setConfiguration(String url)` aus der durch `url` verweisenden XML-Datei[1] gelesen werden. Die Methode `getDatabase()` liefert eine Instanz der Schnittstelle `Database`, über die die Datenbankverbindungen abgewickelt werden.

Database:

Die Schnittstelle `Database` kapselt die JDBC-Verbindung zu der zugrundeliegenden Datenbank und dient als Quelle für Transaktionen. Des weiteren werden Methoden zum Suchen, Löschen, Eintragen und Finden bereitgestellt. Diese Datenbank-Operationen können nur innerhalb von Transaktionen, die durch `begin()` und `commit()` eingeleitet werden, stattfinden. Alle Objekte, die während einer Transaktion in die Datenbank eingetragen bzw. aus dieser geladen werden, gehen in den Zustand `persistent`. Änderungen an persistenten Objekten werden mit der Datenbank synchronisiert, sobald die Transaktion durch `commit()` beendet wird. Wird eine Transaktion erfolgreich oder durch `rollback()` beendet, so gehen diese sich im Speicher befindlichen Objekte in den Zustand `transient`.

Query:

Eine Instanz der Schnittstelle `Query` stellt eine Abfrage an die Datenbank dar, die zur Laufzeit in SQL-Code umgewandelt wird. Durch Aufruf der Methode `execute()` wird die Abfrage an die Datenbank gestartet. Als Ergebnis wird eine Instanz der Schnittstelle `QueryResults` geliefert.

QueryResults:

Eine Instanz dieser Schnittstelle stellt eine Iteration der Ergebnismenge, die durch die zuvor gestartete `Query` geliefert wurde, dar. Diese kann durch die Methode `next()` schrittweise vorwärts durchlaufen werden und liefert das nächste Ergebnisobjekt, welches durch entsprechendes Casting erstellt wird. In Abschnitt 4.3.3.10 folgt hierzu ein Beispiel.

Persistent:

Durch Implementierung der Schnittstelle `Persistent` wird ein Callback-Mechanismus für die Lösch-, Eintrage-, Speicher- und Update-Operationen bereitgestellt.

[1] Eine Erklärung dieser XML-Datei folgt im Abschnitt 5.10

`TimeStampable:`

Objekten, die die Schnittstelle `TimeStampable` implementieren, wird ein Zeitstempel zugewiesen. Dies ist für das optimistische Transaktionskonzept notwendig.

Datenbank-Konfiguration

Es folgt die verwendete XML-Datei `database.xml`, die die Konfigurationsdaten für die Datenbank MySQL beinhaltet:

```
<!DOCTYPE databases PUBLIC "-//EXOLAB/Castor JDO Configuration DTD Version 1.0//EN"
                    "http://castor.exolab.org/jdo-conf.dtd">

<database name="castordiplom" engine="mysql" >
   <driver url="jdbc:mysql://localhost/castordiplom" class-name="com.mysql.jdbc.Driver">
      <param name="user" value="diplom" />
      <param name="password" value="diplom" />
   </driver>
  <mapping href="mapping.xml" />
</database>
```

Dem Wurzelement `database` werden die beiden Elemente `name` und `engine` übergeben. `name` führt die verwendete Datenbank, die unter MySQL zuvor angelegt wurde, ein. Durch das Element `engine` wird der Typ der Datenbank festgelegt. Castor unterstützt eine Vielzahl von Datenbanken; eine Auflistung ist unter [Casto03] zu finden. Innerhalb des Elementes `driver` wird der verwendete Datenbanktreiber durch `class-name` und die URL der Datenbank angegeben. Schließlich werden noch der Benutzername und das zugehörige Passwort bekanntgegeben. Durch den Parameter `href` des Elementes `mapping` wird noch der Verweis auf den Persistenz-Deskriptor eingeführt.

Die Evaluierung wurde nicht nur mit der Datenbank MySQL, sondern auch mit der Datenbank Hypersonic SQL [Hype03] durchgeführt. Die zugehörige Konfigurationsdatei ist auf der beiliegenden Projekt-CD zu finden.

Durchgeführte Änderungen am Prototyp:

Castor unterstützt keine unidirektionalen, sondern nur bidirektionale Beziehungen. Bei den entsprechenden Assoziationen musste wie z.B. zwischen `Primary` und `Secondary`, repräsentiert durch die `ArrayList arrayListP` in der Klasse `Secondary` noch ein Attribut vom Typ `Primary` aufgenommen werden. Dieses wird initialisiert, wenn die `ArrayList` durch die Methode `addArrayListP(Object obj)` mit den entsprechenden Secondary-Objekten gefüllt wird:

```
public void addArrayListP(Object obj)
{
      arrayListP.add(obj);
      ((Secondary) obj).setPrimList(this);
}
```

Diese Vorgehensweise ist für Castor notwendig, da sonst nicht die Fremdschlüssel in der Datenbank initialisiert werden können.

Da die untersuchten Datenbank-Operationen nicht alle innerhalb einer Transaktion, sondern jede Operation innerhalb einer eigenen Transaktion durchgeführt wurde, muss jede Klasse des Prototyps die Schnittstelle `TimeStampable` implementieren. Dies ist notwendig, da die sich im Speicher befindlichen Objekte nach einer zuvor erfolgreich durchgeführten Transaktion den Zustand `transient` besitzen, und erst durch Aufruf der Methode `update` der Schnittstelle `Database` den Zustand `persistent` erhalten und somit innerhalb einer weiteren Transaktion zur Verfügung stehen.

Die Klassen `AnotherClass`, `Primary`, `PrimarySecond` und `Secondary` implementieren noch die Schnittstelle `Persistent`, da Castor nicht automatisch das kaskadierende Löschen unterstützt, d.h.: durch automatischen Aufruf der Methode `jdoAfterRemove()` werden die von diesen Klassen aus referenzierten Objekte manuell gelöscht.

Da die in der Klasse `Primary` befindlichen `java.util.Properties` nicht direkt durch Castor unterstützt werden, wurde eine Hilfsklasse `Property` erstellt. Objekte dieser Klasse besitzen als Attribute den Schlüssel und den zugehörigen Wert. Eine weitere Hilfsklasse `PropertyHelper` übernimmt durch statische Methoden das entsprechende Mapping zwischen den in der Klasse `Primary` befindlichen `java.util.Properties` und der Klasse `Property`. Eine detailliertere Erläuterung dieser Methoden ist in der beiliegenden Java-Doc zu finden.

In der beiliegenden Java-Doc wird die Klasse `CastorTest`, die die verschiedenen Tests durchführt, erläutert.

Vor Ausführung der Tests muss noch eine Datenbank names `castordiplom` in MySQL bzw. HSQL erstellt und die entsprechenden Tabellen in dieser angelegt werden (siehe Abschnitte 6.4 bzw. 6.5). Im nächsten Abschnitt werden die Beziehungen der angelegten Tabellen anhand eines Entity-Relationship-Modells erläutert.

3.2.3.2Entity-Relationship-Modell

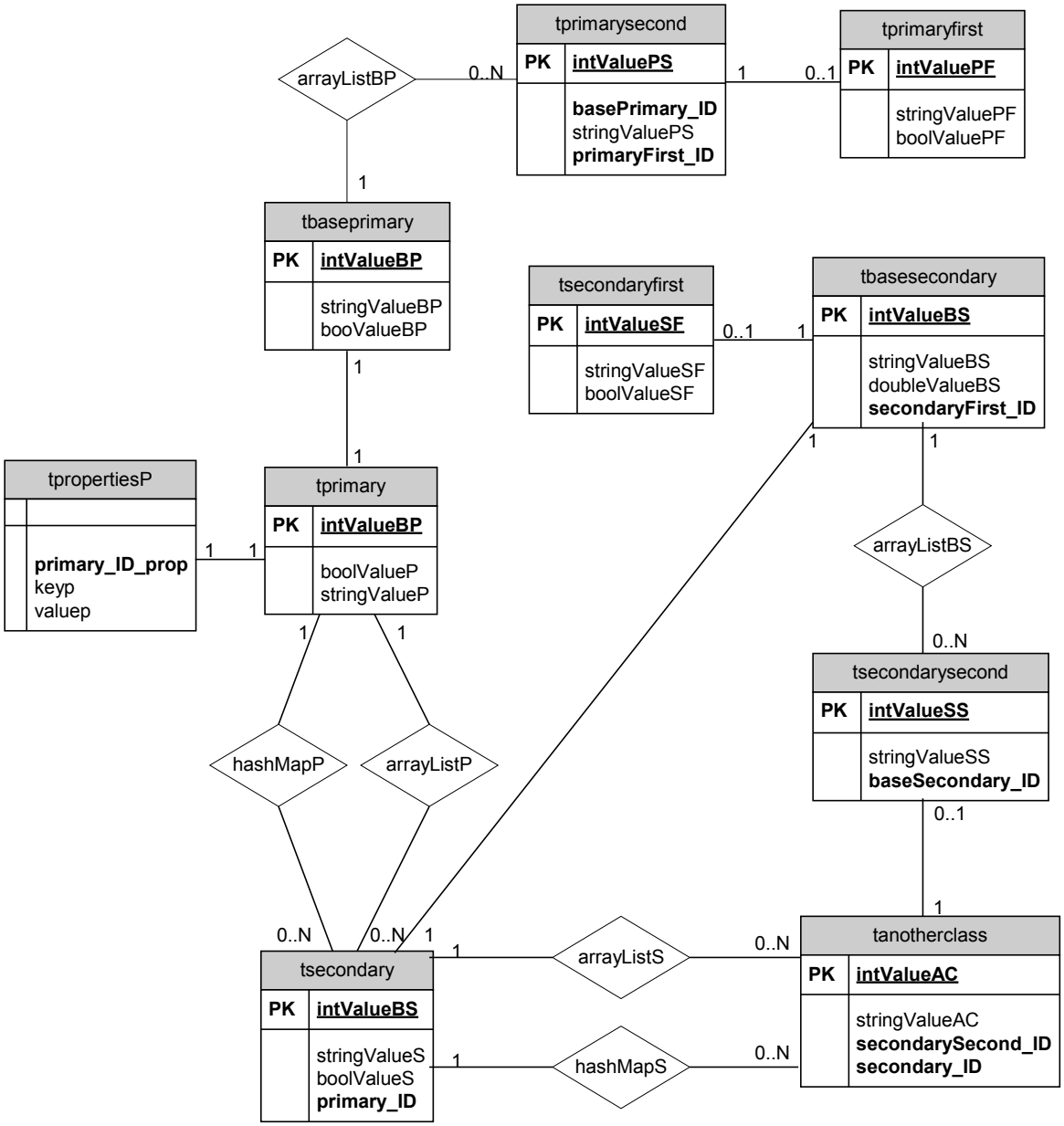


Abbildung 10„Entity-Relationship-Modell der Datenbank castordiplom“

Im obigen Modell sind die **Primärschlüssel** fett-unterstrichen und die **Fremdschlüssel** fett dargestellt. Bei Castor JDO wurde nicht der Serialisierungsmechanismus für die `java.util.Properties` verwendet, sondern diese wurden in einer eigenen Tabelle abgebildet, so dass gezielt nach einem Schlüssel gesucht werden kann.

Der Fremdschlüssel für die `ArrayList` und `HashMap` der Klassen `Primary` und `Secondary` wird jeweils auf eine Tabellenspalte abgebildet: `primary_ID` und `secondary_ID`.

Die Schlüssel der verwendeten `HashMaps` werden nicht auf eine eigene Tabellenspalte abgebildet, da unter Verwendung von Castor JDO, kein eigener Schlüssel für die `HashMap` vergeben werden kann. Es wird automatisch die `ObjektID` des sich in der HashMap befindlichen Objektes als Schlüssel der `HashMap` verwendet.

3.2.3.3Persistenz-Deskriptor

Im Anhang 6.7 ist der erstellte Persistenz-Deskriptor „mapping.xml“ zu finden. Es folgt eine Erläuterung ausgewählter Elemente:

```
<class name="castor.PrimarySecond" identity="intValuePS" depends="BasePrimary">
    <map-to table="tprimarysecond"/>
        <cache-type type="unlimited"/>
    <field name="intValuePS" type="integer" required="true">
        <sql name="intValuePS" type="integer"/>
    </field>
    <field name="stringValuePS" type="string">
        <sql name="stringValuePS" type="varchar"/>
    </field>
    <field name="primaryFirst" type="castor.PrimaryFirst">
        <sql name="primaryFirst_ID"/>
    </field>
    <field name="bp" type="castor.BasePrimary">
        <sql name="basePrimary_ID"/>
    </field>
</class>
```

Durch den Parameter `name` des Elementes `<class>` wird die zu speichernde Klasse bekanntgegeben und durch Angabe des Parameters `identity` wird der PrimaryKey definiert. Der Parameter `depends` sagt aus, dass `BasePrimary` als Master für die abhängige Klasse `PrimarySecond` dient. Das Element `<map-to table>` bezeichnet die Tabelle, auf die die Klasse abgebildet wird. Die Grundeinstellung des Caches ist: `count-limit` mit einer Kapazität von 100, d.h. es werden 100 Objekte dieser Klasse im Cache zwischengespeichert. Dies ist für den untersuchten Anwendungsfall zu wenig, daher wurde ein unbegrenzter Cache „`unlimited`“ verwendet.

Innerhalb des Elementes `<field>` wird beschrieben wie die einzelnen Attribute der Klasse, angegeben durch den Parameter `name`, auf die entsprechende Spalte, durch Element `<sql>` mit Parameter `name` in der Tabelle abgebildet werden. Das Attribut `bp` vom Typ `castor.BasePrimary` wird auf die gleiche Spalte, wie auch der Fremdschlüssel der `arrayListBP` der Klasse `BasePrimary`, abgebildet. Dies wird im nächsten Teilausschnitt des Persistenz-Deskriptor erläutert.

```
<class name="castor.BasePrimary" identity="intValueBP">
    <map-to table="tbaseprimary"/>
        ……
    <field name="arrayListBP" type="castor.PrimarySecond" collection="arraylist">
        <sql many-key="basePrimary_ID"/>
    </field>
</class>
```

Die Klasse `BasePrimary` besitzt eine `ArrayList arrayListBP`, in der Objekte vom Typ `castor.PrimarySecond` verwaltet werden. Die Abbildung der Datentypen `ArrayList`, `HashMap`, `Vector`, `Set`, usw. erfolgt durch Angabe des Parameters `collection`. Unter [Cast03] ist eine Auflistung der Typangaben zu finden.

Da Castor bidirektionale Beziehungen zwischen den Klassen erwartet, muss durch Angabe des Parameters `many-key` des Elementes `<sql>` die gleiche Tabellenspalte, wie auch bei der zugehörigen Objektreferenz angegeben werden. D.h.: da die Klasse `PrimarySecond` eine Objektreferenz `bp` auf die Klasse `BasePrimary` besitzt und die in der `ArrayList arrayListBP` gespeicherten Elemente vom Datentyp

`PrimarySecond` sind, wird der Fremdschlüssel auf die gleiche Tabellenspalte `basePrimary_Id` abgebildet.

```
<class name="castor.Primary" extends="castor.BasePrimary" identity="intValueBP">
    <map-to table="tprimary"/>
        ...........
    <field name="hashMapP" type="castor.Secondary" collection="map">
        <sql many-key="primary_ID_list"/>
    </field>
    <field name="propertiesP" type="castor.Property" collection="collection">
        <sql many-key="primary_ID_prop"/>
    </field>
</class>
```

Innerhalb des Elementes `<class>` wird durch den Parameter `extends` die Vererbung definiert. In obigem Beispiel bedeutet dies, dass `Primary` die Oberklasse `BasePrimary` erweitert. Die Abbildung der Datentypen `Properties` und `Map` erfolgt nach den zuvor für die `ArrayList` erläuterten Regeln.

3.2.3.4Installation + Benutzeranleitung

3.2.3.4.1 Systemvoraussetzungen Software

1. MySQL bzw. HSQL muss installiert, jeweils eine Datenbank `castordiplom` und die zugehörigen Tabellen angelegt worden sein (Anleitung siehe Abschnitte 6.4 bzw. 6.5).

2. Castor JDO liegt als eine Sammlung von JAR-Files vor, die alle in den CLASSPATH aufgenommen werden müssen:

Der CLASSPATH muss folgende JAR-Files beinhalten:

%castorHOME% = c:\castor-0.9.4.3\lib\

%castorHOME%castor-0.9.4.3.jar;
%castorHOME%adaptx_0.9.4.6.jar;
%castorHOME%ant_1.5.jar;
%castorHOME%jakarta-oro-2.0.5.jar;
%castorHOME%jakarta-regexp-1.1.jar;
%castorHOME%jdbc-se2.0.jar;
%castorHOME%jndi_1.2.1.jar;
%castorHOME%jta1.0.1.jar;
%castorHOME%jtf-0.1.jar;
%castorHOME%junit_3.5.jar;
%castorHOME%ldapjdk_4.1.jar;
%castorHOME%postgresql.jar;
%castorHOME%servlet.jar;
%castorHOME%xerces-J_1.4.0.jar;

c:\castor\JDBC\mysql-connector-java-3.0.jar; [Connector/J V3.0] bzw.
c:\castor\hsqldb\lib\hsqldb.jar [Treiber für HSQL-Datenbank]

3. Auch die Konfigurationsdateien „`database.xml`“ und „`mapping.xml`“ müssen in den CLASSPATH aufgenommen werden.

3.2.3.4.2 Systemvoraussetzungen Hardware

Die Tests wurden auf dem im Abschnitt 3.1 genannten Rechner ausgeführt.

3.2.4Implementierung Solarmetric Kodo JDO V2.3.4

Für die Evaluierung von Kodo wurde der erstellte Prototyp von Castor JDO verwendet, da auch Kodo das Konzept der bidirektionalen Beziehungen zwischen den Objekten verwendet (siehe Abschnitt 3.2.3).

Die von dem Castor-Prototyp implementierten Schnittstellen `TimeStampable` und `Persistent` (siehe Abschnitt 3.2.3.1) stehen bei der Kodo-API nicht zur Verfügung.

Da Kodo das kaskadierende Löschen unterstützt, müssen keine weiteren Schnittstellen implementiert werden.

Die Klassen `BasePrimary` und `BaseSecondary` verwenden nun den primitiven Datentyp `int` als Datentyp für die ObjektID (`intValueXX`), da bei Kodo ein eigener Schlüssel bei Verwendung des Datentyp `HashMap` angegeben werden kann. Somit entfällt, dass der Schlüssel der `HashMap` den gleichen Datentyp wie auch die ObjektID der Klasse verwenden muss.

Der Datentyp `Properties` wird direkt durch die JDO-Implementierung unterstützt, daher werden die Hilfsklassen `Property` und `PropertyHelper` des Castor-Prototyp nicht benötigt.

Da das eigene Identitätenkonzept verwendet wurde, wurden die zugehörigen Primary-Key-Klassen vom intelliBO-Prototyp übernommen.

Die erstellte Hilfsklasse `ConfJDO` stellt statische Methoden zur Verfügung. Diese lesen die Konfigurationsdatei `kodo.properties` aus und initialisieren somit die Schnittstelle `PersistenceManagerFactory` und stellen der Anwendung einen `PersistenceManager` zur Verfügung. Die Testklasse `KodoTest` bezieht von der Hilfsklasse `ConfJDO` einen `PersistenceManager` und stellt Methoden für die insert-, delete-, query- und update- Operationen zur Verfügung. In der beiliegenden Java-Doc werden die erstellten Klassen und deren Methoden erläutert. Weiterhin werden im Abschnitt 2.2.2 alle verwendeten Klassen bzw. Schnittstellen der JDO-Spezifikation erklärt.

Vor Ausführung der Tests muss noch eine Datenbank names `kodoDiplom` in MySQL bzw. HSQL erstellt und die entsprechenden Tabellen in dieser angelegt werden. Das SQL-Skript zum Anlegen der Tabellen ist im Anhang 6.11 zu finden. Im nächsten Abschnitt werden die Beziehungen der angelegten Tabellen anhand eines Entity-Relationship-Modells dargestellt.

Die durch die Hilfsklasse `ConfJDO` ausgelesene Konfigurationsdatei **kodo.properties** wird im folgenden beschrieben:

Implementierung der PersistenceManagerFactory der Kodo-API
javax.jdo.PersistenceManagerFactoryClass=com.solarmetric.kodo.impl.jdbc.JDBCPersistenceManagerFactory

durch false wird das pessimistische Transaktionskonzept eingeschaltet
javax.jdo.option.Optimistic=false

false: nach commit werden gelesenen Werte aus dem Speicher entfernt

javax.jdo.option.RetainValues=false

true: nach rollback werden die Werte wieder auf den ursprünglichen Wert gesetzt
javax.jdo.option.RestoreValues=true

true: Daten können ausserhalb einer Transaktion gelesen werden
javax.jdo.option.NontransactionalRead=false

true: Daten können ausserhalb einer Transaktion geschrieben werden
javax.jdo.option.NontransactionalWrite=false

Datenbank-URL unter Verwendung von MySQL
javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/kodoDiplom

Datenbank-URL unter Verwendung von HSQL im Server-Modus
javax.jdo.option.ConnectionURL=jdbc:hsqldb:hsql://localhost

Datenbank-URL unter Verwendung von HSQL im embedded-Modus
javax.jdo.option.ConnectionURL=jdbc:hsqldb:hsqlkodo

Benutzername für die Datenbank
javax.jdo.option.ConnectionUserName= diplom

Password der Datenbank
javax.jdo.option.ConnectionPassword= diplom

JDBC-Datenbanktreiber Connector/J für MySQL
javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver

JDBC-Datenbanktreiber für HSQL
javax.jdo.option.ConnectionDriverName=org.hsqldb.jdbcDriver

default = 30; -1 schaltet lazy loading[1] aus
com.solarmetric.kodo.DefaultFetchThreshold=-1

default = 10; wieviele Zeilen vorgefetcht werden
com.solarmetric.kodo.DefaultFetchBatchSize=10

default = false; erlaubt die Verwendung von extensions bei Query-Abfragen
com.solarmetric.kodo.EnableQueryExtensions=false

EvaluierungsLizenzSchlüssel; Schlüssel ist nur 30 Tage bis zum 15.Mai gültig
com.solarmetric.kodo.LicenseKey=0333-A135-A308-0919-0900

spezifiziert die Implementierung des PersistenceManagers
com.solarmetric.kodo.PersistenceManagerClass=com.solarmetric.kodo.runtime.PersistenceManagerImpl

default: false; Ob Soft-Referenzen verwendet werden sollen; true bringt bei vielen Objekte
Performance-Vorteile
com.solarmetric.kodo.UseSoftTransactionCache=true

Datei die bestimme Mapping-Informationen der Datentypen für die verwendete Datenbank besitzt
Mapping für MySQL-Datenbank
com.solarmetric.kodo.impl.jdbc.DictionaryClass=com.solarmetric.kodo.impl.jdbc.schema.dict.MySQLDictionary

Mapping für HSQL-Datenbank
##com.solarmetric.kodo.impl.jdbc.DictionaryClass=com.solarmetric.kodo.impl.jdbc.schema.dict.HSQLDictionary

spezielle Eigenschaften der verwendeten Datenbank, z.B.: Tabellentyp innoDB bei MySQL
com.solarmetric.kodo.impl.jdbc.DictionaryProperties=TableType=innoDB,SupportsSelectForUpdate=true

default: true; false sagt aus, dass alle an einer Vererbung beteiligten Klassen in einer eigenen
Tabelle abgebildet werden
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping=false

[1] Ein durch „lazy loading“ aufgetretenes Problem wird in Abschnitt 4.4.5.3 erläutert

Obige Einstellungen der Konfigurationsparameter ergaben die besten Performance-Ergebnisse.

3.2.4.1 Entity-Relationship-Modell

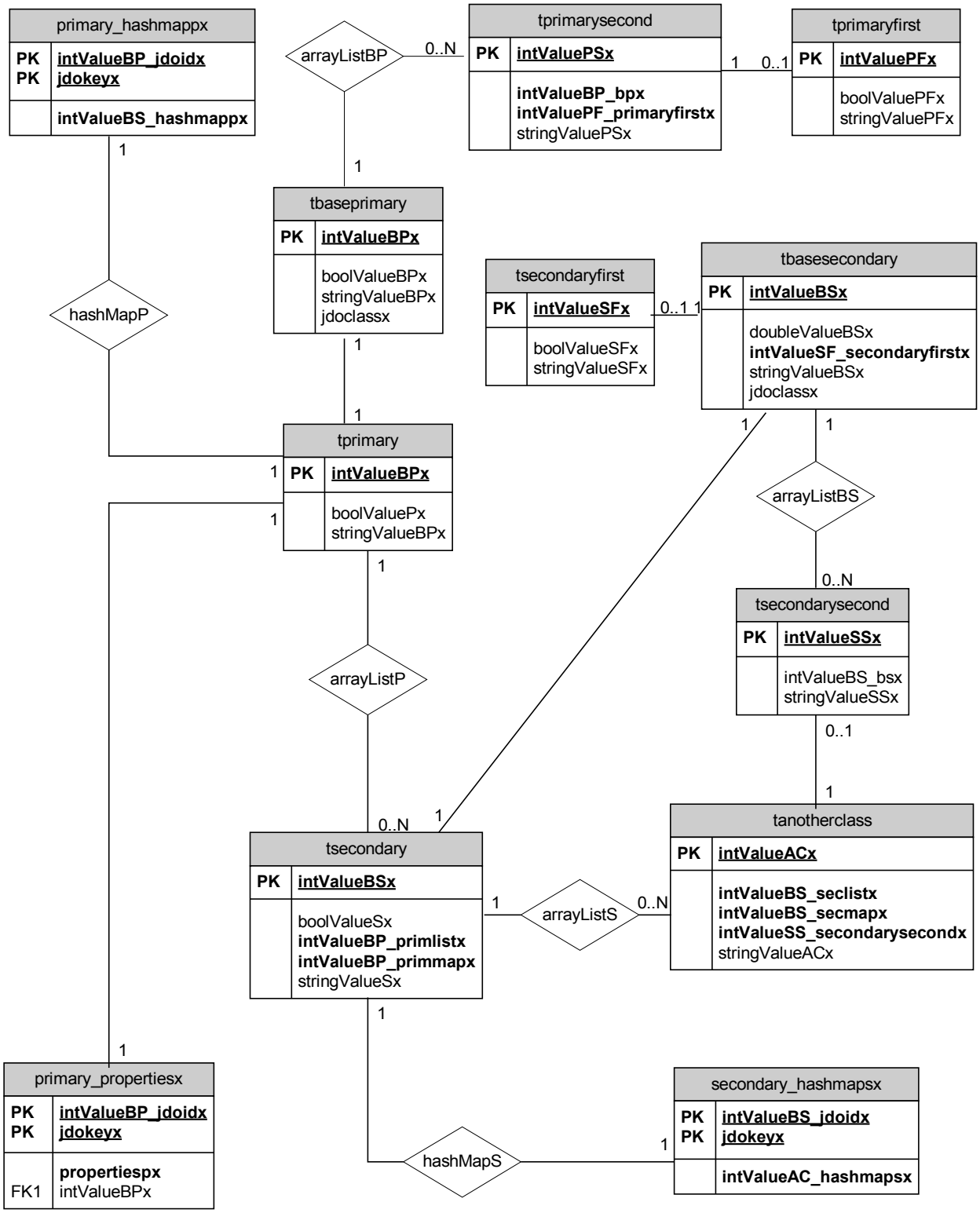


Abbildung 11„Entity-Relationship-Modell der Datenbank kodoDiplom“

Im obigen Modell sind die **Primärschlüssel** fett-unterstrichen und die **Fremdschlüssel** fett dargestellt. Der Datentyp `Properties` wird in der Tabelle `primary_propertiesx` abgebildet. Der Schlüssel der `Properties` wird in der Spalte `jdokeyx` und der zugehörige Wert in der Spalte `propertiespx` verwaltet. Über den Fremdschlüssel `intValueBP_jdoidx` wird die Beziehung zu der Tabelle `tprimary` hergestellt.

Weitherhin ist zu erkennen, dass Kodo für die Verwendung des Datentyp `HashMap` – Bestandteil der Klassen `Primary` und `Secondary` – jeweils eine eigene Tabelle erzeugt. In der Tabelle `primary_hashmappx` wird der Schlüssel der `HashMap` in der Spalte `jdokeyx` und der zugehörige Wert über den Fremdschlüssel `intValueBS_hashmappx` verwaltet. Über den Fremdschlüssel `intValueBP_jdoidx` wird die Beziehung zu der Tabelle `tprimary` hergestellt.

Die Tabellen `tbaseprimary` und `tbasesecondary` verwalten jeweils noch eine Spalte `jdoclassx`, in der der vollständige Klassenname der Unterklassen verwaltet wird.

Die Fremdschlüssel für die `ArrayList` und `HashMap` werden jeweils in einer eigenen Spalte abgebildet.

3.2.4.2Persistenz-Deskriptor

Im Anhang 6.10 ist der erstellte Persistenz-Deskriptor „`kododiplom.jdo`“ zu finden. Es folgt eine Erläuterung ausgewählter Elemente:

```
<package name="kododiplom">
<class identity-type="application" name="AnotherClass" objectid-class="AnotherClassKey" requires-
extent="true">
    <extension vendor-name="kodo" key="table" value="tanotherclass"/>
    <extension vendor-name="kodo" key="lock-column" value="none"/>
    <extension vendor-name="kodo" key="class-column" value="none"/>
    <field name="secondarySecond">
        <extension vendor-name="kodo" key="dependent" value="true"/>
    </field>
    <field name="intValueAC" primary-key="true"/>
</class>
```

Durch das Attribut `name` des Elementes `<package>` wird der Name des Package, in welchem sich die Klasse befindet eingeführt.

Die Parameter `identity-type`, `name` und `objectid-class` des Elementes `<class>` sagen aus, dass die Klasse `AnotherClass` durch das eigene Identitätenkonzept (`identity-type="application"`) verwaltet wird. Die Primary-Key-Klasse `AnotherClassKey` übernimmt dies. Durch den Bytecode-Enhancement-Vorgang wird dieser Klasse automatisch die Schnittstelle `PersistenceCapable` hinzugefügt. Durch setzen von `requires-extent="true"`, ist es möglich über einen `Extent`[1] auf diese Klasse zuzugreifen.

Durch das Attribut `vendor-name="kodo"` des Elementes `<extension>` werden KODO-spezifische Konfigurationen vorgenommen. Dies geschieht als Schlüssel- (`key`) Wert- (`value`) Paar. Der Schlüssel `table` sagt aus, dass die Klasse `AnotherClass` auf die Tabelle `tanotherclass` abgebildet wird.

[1] Die Klasse Extent wurde im Abschnitt 2.2.2 eingeführt

Durch den Schlüssel `lock-column` wird die Spalte angegeben, in der eine Versionsnummer der geänderten Objekte verwaltet wird. Dies ist bei Verwendung von optimistischen Transaktionen notwendig, um konkurrierende Veränderungen an Objekten zu erkennen. Da bei der Evaluierung das pessimistische Transaktionskonzept verwendet wurde, wurde die Versionsverwaltung durch `none` explizit ausgeschaltet. Geschieht dies nicht, versucht Kodo die Versionsnummer in der Spalte `jdolockx` zu speichern. Dies führt zu einer Fehlermeldung, falls sich diese Spalte nicht in der zugehörigen Tabelle befindet.

Der Schlüssel `class-column` spezifiziert die Spalte in der Tabelle, in der der vollständige Klassenname des Objektes gespeichert wird. Da die Klasse `AnotherClass` keine Oberklasse darstellt, wird diese Option nicht benötigt und somit durch `none` deaktiviert.

Durch das Element `<field>` werden die persistenten Attribute einer Klasse beschrieben. Die JDO-Spezifikation definiert Regeln, die die persistenten Felder bestimmter Datentypen einer Klasse automatisch erkennen. Daher muss im zugehörigenen Persistenz-Deskriptor kein explizites Mapping dieser Datentypen erfolgen. Diese Regeln sind in [Russ02, S. 125] zu finden.

Das Attribut `secondarysecond` der Klasse `AnotherClass` repräsentiert eine 1...1-Beziehung zu der Klasse `SecondarySecond`. Diese 1...1-Beziehung wird im Persistenz-Deskriptor durch Angabe des Schlüssels `dependent` auf `true` ausgedrückt. Weiterhin wird durch diese Angabe das kaskadierende Löschen aktiviert, d.h. wenn das `AnotherClass`-Objekt gelöscht wird, wird auch automatisch das von diesem aus referenzierte `SecondarySecond`-Objekt mitgelöscht[1].

Durch Setzen von `primary-key` auf `true` wird ausgedrückt, dass das Attribut `intValueAC` als PrimaryKey der Klasse fungiert.

```
<class identity-type="application" name="Primary" persistence-capable-superclass="BasePrimary" requires-extent="true">
.....
    <field name="hashMapP">
        <map key-type="java.lang.Integer" value-type="kododiplom.Secondary"/>
            <extension vendor-name="kodo" key="value-dependent" value="true"/>
            <extension vendor-name="kodo" key="key-dependent" value="true"/>
            <extension vendor-name="kodo" key="inverse" value="primMap"/>
    </field>
.....
</class>
```

Durch Angabe des Parameter `persistence-capable-superclass` wird die Oberklasse dieser Klasse bekannt gegeben. Unter Verwendung des eigenen Identitätenkonzeptes erben somit auch die Unterklassen die PrimaryKey-Klasse der Oberklasse.
Das Attribut `hashMapP` der Klasse `Primary` ist vom Datentyp `HashMap`. Dieser implementiert die Schnittstelle `Map`, und wird durch das Element `<map>` beschrieben. Innerhalb dieses Elementes werden durch die Parameter `key-type` und `value-type` die verwendeten Datentypen der Schlüssel bzw. zugehörigen Werte der `HashMap` bekannt gegeben. Über die Schlüssel/Wert-Paare `value-dependent` und `key-`

[1] In der Datenbank wird das kaskadierende Löschen über die Fremdschlüsselbeziehungen zwischen den beteiligten Tabellen erreicht.

`dependent` jeweils `true` des Elementes `<extension>` wird das kaskadierende Löschen für die Schlüssel und Werte der `HashMap` aktiviert.
Durch den Schlüssel `inverse` wird die bidirektionale Beziehung zwischen den Klassen `Primary` und `Secondary` eingeführt. Im obigen Beispiel bedeutet dies, dass die Klasse `Secondary` ein Attribut `primMap` des Datentyp `Primary` besitzt, welches initialisiert wird, wenn die `HashMap` durch die Methode `addHashMapP(Object obj)` mit den entsprechenden `Secondary`-Objekten gefüllt wird:

```
public void addHashMapP(Object obj)
{
    hashMapP.put(new Integer( ((Secondary) obj).getIntValueBS()), obj);
    ((Secondary) obj).setPrimMap(this);
}
```

Im folgenden wird das Mapping einer `ArrayList` erläutert:

```
<field name="arrayListP">
    <collection element-type="kododiplom.Secondary"/>
        <extension vendor-name="kodo" key="inverse" value="primList"/>
        <extension vendor-name="kodo" key="element-dependent" value="true"/>
</field>
```

Durch das Attribut `element-type` des Elementes `<collection>` (`ArrayList` implementiert die Schnittstelle `Collection`) wird der Datentyp der in der `ArrayList` gespeicherten Objekte gesetzt. Auch hier wird durch das Attribut `inverse` die bidirektionale Beziehung zwischen den Klassen `Primary` und `Secondary` bekannt gegeben. Durch Setzen des Attributes `element-dependent` auf `true` wird das kaskadierende Löschen aktiviert.

```
<field name="propertiesP" persistence-modifier="persistent">
    <map key-type="java.lang.String" value-type="java.lang.String"/>
        <extension vendor-name="kodo" key="value-dependent" value="true"/>
        <extension vendor-name="kodo" key="key-dependent" value="true"/>
</field>
</class>
```

Da das Attribut `propertiesP` vom Datentyp `Properties` ist, und dieses die Schnittstelle `Map` implementiert, werden auch hier die Datentypen der Schlüssel- und Werte der `Properties` bekannt gegeben. Obwohl die Datentypen der Schlüssel und zugehörigen Werte des Datentyp `Properties` immer vom Datentyp `String` sind, muss dies explizit im Persistenz-Deskriptor angegeben werden.

3.2.4.3Installation + Benutzeranleitung

3.2.4.3.1 Systemvoraussetzungen Software

1. MySQL bzw. HSQL müssen installiert, jeweils eine Datenbank `kododiplom` und die zugehörigen Tabellen angelegt worden sein (Anleitung siehe Abschnitt 6.4 bzw. 6.5).

2. Kodo JDO liegt als eine Sammlung von JAR-Files vor, die alle in den CLASSPATH aufgenommen werden müssen:

Der CLASSPATH muss folgende JAR-Files beinhalten:

%kodoHOME% = c:\kodo-jdo-2.4.3\lib\

Erklärung der JAR-Files in der Kodo-Dokumentation

%kodoHOME%jakarta-commons-logging-1.0.2.jar;	[Bibliothek für Logging]
%kodoHOME%jca1.0.jar;	[Java Connector Architecture]
%kodoHOME%jdbc2_0-stdext.jar;	[JDBC 2.0 Erweiterungen]
%kodoHOME%jdo1_0.jar;	[JDO-Spezifikation]
%kodoHOME%jndi.jar;	[JNDI-Bibliothek]
%kodoHOME%jta-spec1_0_1.jar;	[Transaktionssteuerung]
%kodoHOME%kodo-jdo.jar;	[Kodo JDO Kern]
%kodoHOME%kodo-jdo-runtime.jar;	[Kodo JDO Laufzeit Kern]
%kodoHOME%kodo-reverse-schema.jar;	[Bibliothek für das Schema-Tool]
%kodoHOME%log4j-1.2.6.jar;	[Bibliothek für Logging]
%kodoHOME%serp.jar;	[interne Hilfsklasse für Kodo]
%kodoHOME%xml-apis.jar;	[APIs für XML-Parsing]
c:\kododiplom\JDBC\mysql-connector-java-3.0.jar;	[Connector/J V3.0] bzw.
c:\kododiplom\hsqldb\lib\hsqldb.jar	[Treiber für HSQL-Datenbank]

3. Auch die Konfigurationsdateien „`kododiplom.jdo`“, „`kodo.properties`“ und „`log4j.properties`“ müssen in den CLASSPATH aufgenommen werden.

Die Konfigurationsdatei „`log4j.properties`[1]“ ist notwendig, da bei Kodo alle automatisch erzeugten SQL-Statements durch die log4j-API am Bildschirm ausgegeben werden. Es gibt keine Einstellmöglichkeit in der Konfigurationsdatei „`kodo.properties`“, mit der dieses Logging ausgeschaltet werden kann.

3.2.4.3.2 Systemvoraussetzungen Hardware

Die Tests wurden auf dem im Abschnitt 3.1 genannten Rechner ausgeführt.

3.2.5Implementierung Libelis Lido V1.4

Für die Evaluierung von Lido wurde der erstellte Prototyp von Kodo verwendet, da auch Lido das Konzept der bidirektionalen Beziehungen zwischen den Objekten verwendet (siehe Abschnitt 3.2.4).

Lediglich implementieren die Klassen `AnotherClass`, `Primary`, `PrimarySecond` und `Secondary` die Schnittstelle `InstanceCallbacks`, da Lido nicht automatisch das kaskadierende Löschen unterstützt.

Da das eigene Identitätenkonzept verwendet wurde, wurden auch die zuvor erstellten Primary-Key-Klassen übernommen.

Die erstellte Hilfsklasse `ConfJDO` stellt statische Methoden zur Verfügung. Diese lesen die Konfigurationsdatei „`lido.properties`“ aus und initialisieren somit die

[1] Über diese Konfigurationsdatei wird die automatische Ausgabe der erstellten SQL-Statemens unterbunden. Es wird nur zugelassen, dass bei Auftreten eines Error eine Ausgabe am Bildschirm erfolgt. Das Abschalten ist notwendig, da sonst die Ausgabe am Bildschirm die Performance-Tests beinflusst.

Schnittstelle `PersistenceManagerFactory` und stellen der Anwendung einen `PersistenceManager` zur Verfügung.

Die Testklasse `LidoTest` bezieht von der Hilfsklasse `ConfJDO` einen `PersistenceManager` und stellt Methoden für die insert-, delete-, query- und update-Operationen zur Verfügung. In der beiliegenden Java-Doc werden die erstellten Klassen und deren Methoden erläutert. Weitherhin werden im Abschnitt 2.2.2 alle verwendeten Klassen und Schnittstellen der JDO-Spezifikation erklärt.

Vor Ausführung der Tests muss noch eine Datenbank names `lidoDiplom` in MySQL bzw. HSQL erstellt und die entsprechenden Tabellen angelegt werden. Das SQL-Skript zum Anlegen der Tabellen ist in Abschnitt 6.13 zu finden. Im nächsten Abschnitt werden die Beziehungen der angelegten Tabellen anhand eines Entity-Relationship-Modells dargestellt.

Es folgt eine Eräuterung der durch die HIlfsklasse `ConfJDO` ausgelesenen Konfigurationsdatei „`lido.properties`“. Es werden nur die Lido-spezifischen Konfigurationen dargestellt[1]:

tracing (Fehlerprotokollierung) ist per default ausgeschaltet
lido.trace=PM,CACHE,DUMP,QUERY,SQL,SQLMAPPING,POOLING,SQLQUERY

trace ein oder ausschalten
lido.trace.enabled=false

sagt aus, wieviele Connections schon zu Programmstart geöffnet sind
lido.minPool=0

0 kein Limit von Connections
lido.maxPool=0

ob soft- oder weak-Referenzen von Objekten im Cache gehalten werden
lido.cache.entry-type=soft

Cache-Strategie
lido.cache.mode=local

Obige Einstellungen der Konfigurationsparameter ergaben die besten Performance-Ergebnisse.

[1] Die durch die JDO-Spezifikation definierten Konfigurationen werden in Abschnitt 3.2.4 vorgestellt.

3.2.5.1 Entity-Relationship-Modell

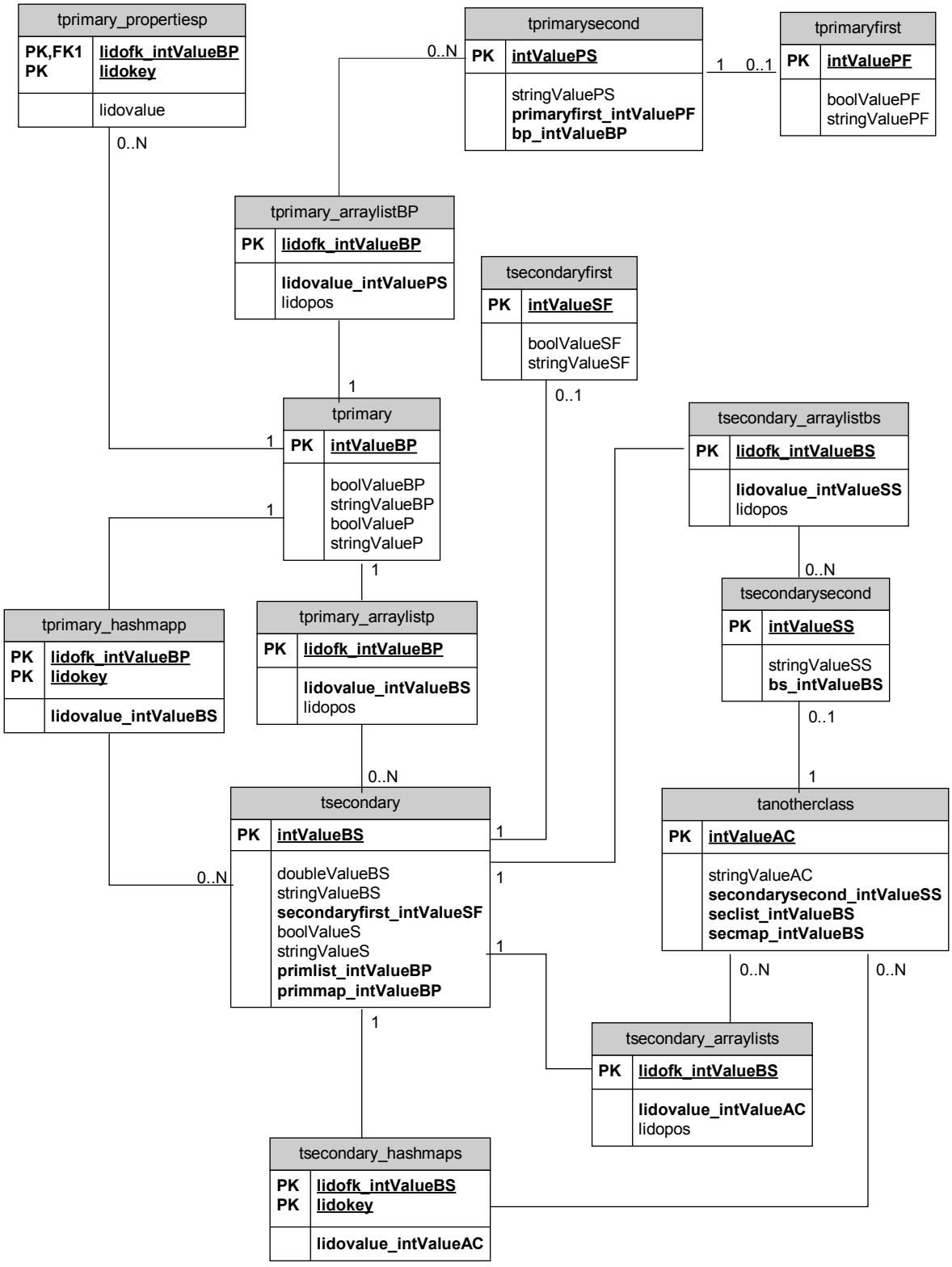


Abbildung 12"Entity-Relationship-Modell der Datenbank lidodiplom"

Im obigen Modell sind die **Primärschlüssel** fett-unterstrichen und die **Fremdschlüssel** fett dargestellt. Der Datentyp `Properties` wird in der Tabelle `tprimary_properties` abgebildet. Der Schlüssel der `Properties` wird in der Spalte `lidokey` und der zugehörige Wert in der Spalte `lidoValue` verwaltet. Über

den Fremdschlüssel `lidofk_intValueBP` wird die Beziehung zu der Tabelle `tprimary` hergestellt.

Bei Lido werden alle an der Vererbung beteiligten Klassen in einer Tabelle abgebildet, d.h. in der Tabelle der am weitesten abgeleiteten Klasse werden alle Attributwerte mitabgespeichert.

Bei Verwendung der Datentypen `List` und `Map` wird jeweils eine eigene Tabelle benötigt. In der Tabelle, die die `List` repräsentiert werden die beiden Fremdschlüssel und die Position `lidopos` des Elementes in der `List` gespeichert. Die Tabelle, die die `Map` darstellt, wird der Schlüssel der `Map` und die beiden Fremdschlüssel abgelegt.

Die Fremdschlüssel für die `ArrayList` und `HashMap` werden jeweils in einer eigenen Spalte abgebildet.

3.2.5.2Persistenz-Deskriptor

In Abschnitt 6.12 ist der erstellte Persistenz-Deskriptor „`lidodiplom.jdo`“ zu finden. Es folgt eine Erläuterung von Lido-spezifischen Elementen:

```
.....
<extension vendor-name="libelis" key="sql-name" value="tanotherclass"/>
<field name="intValueAC" primary-key="true"/>
<field name="secondarySecond">
</field>
.....
```

Eine 1…1-Beziehung wird automatisch durch Lido erkannt. Es muss lediglich der Attributbezeichner durch den Paramter `name` des Elementes `<field>` eingeführt werden.

```
.....
<field name="arrayListBP">
<collection element-type="lidodiplom.PrimarySecond"/>
<extension vendor-name="libelis" key="collection-type" value="java.util.ArrayList"/>
<extension vendor-name="libelis" key="sql-reverse" value="javaField:bp"/>
</field>
.....
```

Eine `ArrayList`, diese implementiert die Schnittstelle `Collection`, wird Lido durch das Element `<collection>` bekannt gemacht. Durch den Parameter `element-type` wird der Datentyp der Elemente der Liste festgelegt. Der Parameter `vendor-name="libelis"` des Elementes `<extension>` leitet die Lido-spezifischen Abbildungsregeln durch Schlüssel(`key`) / Wert(`value`)-Paare ein. Der Schlüssel `collection-type` spezifiziert die Klasse, die die Schnittstelle `Collection` implementiert. Lido unterstützt bidirektionale Beziehungen (Erklärung siehe Abschnitt 3.2.4.2) zwischen Klassen. Dies wird durch den Schlüssel `sql-reverse` eingeführt. Der zugehörige Wert drückt aus, dass in der Klasse `PrimarySecond` ein Attribut vom Datentyp `BasePrimary` in der Klasse `PrimarySecond` vorhanden ist.

```
.....
<field name="hashMapP">
<map key-type="java.lang.Integer" value-type="lidodiplom.Secondary"/>
<extension vendor-name="libelis" key="map-type" value="java.util.HashMap"/>
<extension vendor-name="libelis" key="sql-reverse" value="javaField:primMap"/>
</field>
.....
```

Durch das Element `<map>` wird Lido eine `Map` bekannt gemacht. Die Parameter `key-type` und `value-type` deklarieren den Datentyp der Schlüssel und der zugehörigen Werte. Durch den Schlüssel `map-type` des Elementes `<extension>` wird die Klasse, die die Schnittstelle `Map` implementiert eingeführt. Durch `sql-reverse` wird die bidirektionale Beziehung zwischen den Klassen `Primary` und `Secondary` eingeleitet. In obigem Fall bedeutet dies, dass die Klasse `Secondary` ein Attribut `primMap` vom Datentyp `Primary` besitzt.

3.2.5.3Installation + Benutzeranleitung

3.2.5.3.1 Systemvoraussetzungen Software

1. MySQL bzw. HSQL müssen installiert, jeweils eine Datenbank `lidodiplom` und die zugehörigen Tabellen angelegt worden sein (Anleitung siehe Abschnitt 6.4 bzw. 6.5).

2. Lido JDO liegt als eine Sammlung von JAR-Files vor, die alle in den CLASSPATH aufgenommen werden müssen:

Der CLASSPATH muss folgende JAR-Files beinhalten:

%lidoHOME% = c:\dev\Lido\lib\

%lidoHOME%jaas_1_0_0.jar;	[Java Authentication API]
%lidoHOME%jdbc_2_0-stdext.jar;	[JDBC 2.0 Erweiterungen]
%lidoHOME%jdo_1_0_0.jar;	[JDO-Spezifikation]
%lidoHOME%jta_1_0_1.jar;	[Transaktionssteuerung]
%lidoHOME%lido-api.jar;	[Lido-API]
%lidoHOME%lido-dev.jar;	[Entwicklungstools]
%lidoHOME%lido-rdb.jar;	[Unterstützung für RDBMS]
%lidoHOME%lido-rt.jar;	[Lido Kern]
%lidoHOME%servlet.jar;	[Servlet_unterstützung]
%lidoHOME%skinlf.jar;	[Oberflächen-Skins]
c:\dev\Lido\bin;	[Lizenz-Datei]
c:\lidodiplom\JDBC\mysql-connector-java-3.0.jar;	[Connector/J V3.0] bzw.
c:\lidodiplom\hsqldb\lib\hsqldb.jar	[Treiber für HSQL-Datenbank]

3. Auch die Konfigurationsdateien „`lidodiplom.jdo`“ und „`lido.properties`“ müssen in den CLASSPATH aufgenommen werden.

3.2.5.3.2 Systemvoraussetzungen Hardware

Die Tests wurden auf dem im Abschnitt 3.1 genannten Rechner ausgeführt.

4. Ergebnisse und Fazit

4.1 Signsoft intelliBO

In diesem Abschnitt werden die Ergebnisse der Evaluierung von Signsoft intelliBO vorgestellt und auf Probleme, die während der Implementierung auftraten bzw. Probleme von intelliBO aufgelistet. Zu Beginn der Diplomarbeit stand die Version 3.1 von intelliBO als offizielle Version zur Verfügung. Aufgetretene Probleme wurden mit Signsoft besprochen und wurden teilweise in der momentan noch in der Entwicklung befindlichen Version 3.2 behoben. Eine Beta-Version der neuen Version wurde von Signsoft während der Evaluierung zur Verfügung gestellt. Gegenüber der Version 3.1 sind bekannt gewordene Bugs behoben worden und insgesamt soll diese Version performanter als die Version 3.1 sein, da Fremdimplementierungen für die Klassen der Schnittstelle `Collection` verwendet wurden.

Nachdem der Prototyp erstellt worden ist, wurden die zugehörigen Persistenz-Deskriptoren mit Hilfe der graphischen Entwicklungsumgebung von intelliBO erstellt. Diese sind auf der Projekt-CD zu finden. Für eine detaillierte Erklärung zur Erstellung der Persistenz-Deskriptoren wird auf [Sign02] verwiesen.

Als nächstes wurde der Tabellenskript-Generator von intelliBO getestet. Dieser erzeugt anhand der zuvor erstellten Persistenz-Deskriptoren die entsprechenden SQL-Skripte zur Erzeugung der Tabellen für die zugrunde liegende Datenbank.

4.1.1 Automatisches Erstellen der zugehörigen Tabellen

In der graphischen Entwicklungsumgebung besteht nur die Möglichkeit die zugrunde liegende Datenbank anzugeben, sprich den verwendeten JDBC-Datenbanktreiber, die URL, Benutzername und Password der Datenbank und einen Verbindungstest durchzuführen. Leider besteht nicht die Möglichkeit bestimmte Konfigurationsangaben, die für MySQL nötig sind z.B. den verwendeten Tabellentyp, anzugeben. Wird der Tabellenskript-Generator von intelliBO ausgeführt, so werden für jede zu speichernde Klasse ein CREATE TABLE- Statement erzeugt, welches für MySQL jedoch ein paar Fehler aufwies:

Bsp.: erzeugtes SQL-Skript zum Anlegen der Tabelle TPRIMARY:

```
CREATE TABLE TPRIMARY (FINTVALUEBP INT(10), FBOOLVALUEP TINYINT(3), FSTRINGVALUEP VARCHAR(255), FDOUBLEVALUEP DOUBLE (17), FPROPERTIESP VARCHAR(255), PRIMARY KEY (FINTVALUEBP));
```

- Wird bei MySQL ein PrimaryKey angegeben, so muss das zugehörige Attribut, in diesem Fall `FINTVALUEBP` explizit auf `not null`[1] gesetzt werden.

- Ein weiteres Problem bestand darin, dass durch die Standard-Einstellung von intelliBO der Java-Datentyp `double` nicht korrekt auf den SQL-Datentyp `double` abgebildet wird. Entweder wird als SQL-Datentyp nur `double` angegeben, oder die Angabe des Formates muss in der Form (n,m) erfolgen. n bedeutigt die Gesamtstellenanzahl und m die von n beanspruchte Anzahl von Nachkommestellen. Dieses Problem kann behoben werden, wenn in der Datei `dbsupport_mysql.xml` zu finden in dem Verzeichnis C:\Signsoft\intelliBO3\lib

[1] Sagt aus, dass diese Spalte auf jeden Fall einen Wert ungleich NULL beinhalten muss, und durch die Angabe PrimaryKey, dass dieser Wert von allen anderen Werten in dieser Spalte verschieden sein muss

der intelliBO-Installation „von Hand“ Korrekturen an den Abbildungsregeln von den Java-Datentypen auf die SQL-Datentypen vorgenommen werden.

- Da nicht explizit der verwendete Tabellentyp von MySQL, in unserem Fall innoDB, angegeben werden kann, muss auch dies manuell in das CREATE TABLE- Statement aufgenommen werden: `type=innoDB`

- Weiterhin ist zu erkennen, dass der Java-Datentyp `Properties` auf ein SQL-Datentyp `VARCHAR(255)` abgebildet wird. Dies muss manuell auf den SQL-Datentyp `BLOB` geändert werden. In Abschnitt 4.1.5 wird das dadurch resultierende Problem erläutert.

Das dann „von Hand“ korrigierte CREATE TABLE- Statement:

```
CREATE TABLE TPRIMARY (FINTVALUEBP INT(10) not null, FBOOLVALUEP TINYINT(3), FSTRINGVALUEP VARCHAR(255),
FDOUBLEVALUEP DOUBLE (17,5), FPROPERTIESP blob, PRIMARY KEY (FINTVALUEBP)) type=innoDB;
```

Fremdschlüssel-Beziehungen zwischen den einzelnen Tabellen werden in dem jeweiligen Persistence-Deskriptor durch Hilfe des Elementes `<reference>` angegeben. Eine Erklärung hierzu ist im Abschnitt 4.1.5 zu finden.

Bei unseren neun erzeugten Tabellen sind diese Probleme noch akzeptabel „von Hand“ zu beseitigen, aber bei größeren Projekten, bei denen weit über 100 Tabellen zu verwalten sind, kann dies schnell unübersichtlich werden.

4.1.2Integrationsaufwand für bereits existierende Klassen

IntelliBO nutzt einen Bytecode-Enhancer, um den Klassen der Applikation die jdo-spezifischen Methoden zu zuführen, d.h. eine Erweiterung des Objektmodell, sprich die Klassen von jdo-spezifischen Oberklassen erben lassen oder bestimmte Schnittstellen zu implementieren ist nicht nötig. Dies geschieht vollkommen transparent für den Anwender während dem Enhancement-Vorgang. Dieser Vorgang benötigt lediglich die zuvor erstellten Persistenz-Deskriptoren, wertet diese aus und erweitert den Bytecode mit Datenbank- bzw. jdo-spezifischen get- und set-Methoden. Diese Vorgehensweise ist eher nachteilig, da man keine Kontrolle über die erweiterten Klassen hat. Letztendlich ist es Sache des Hersteller der JDO-Implementierung, wie er den Enhancement-Vorgang implementiert. Die Spezifikation schreibt hier keine allgemeine Vorgehensweise voraus. Weitere Möglichkeiten den Enhancement-Vorgang umzusetzen wären: die Nutzung der Java-Reflection-API oder den Source-Code direkt zu erweitern. Des weiteren werden keine SQL-Kenntnisse benötigt und auch keine JDBC-Anweisungen müssen im Programmcode verankert[1] werden.

4.1.3Performance und Ressourcenverbrauch

Zu Beginn wurden n verschiedene Primary-Objekte des Objektmodells in n Dateien serialisiert, diese wieder deserialisiert und in einer `ArrayList` zwischengespeichert. Die Zeiten wurden durch Aufruf des Befehls `System.currentTimeMillis()` gemessen. Der Speicherverbrauch wurde im Ressourcenfenster des Taskmanagers abgelesen. Dies diente dazu einen Vergleich für die `insert`- und `query`- Anweisungen durch JDO zu erhalten. Es wurden jeweils 5 Tests mit 100, 500 und 1000

[1] Wenn die Tabellen nicht explizit „von Hand“ angelegt werden sollen, kann dies auch durch JDBC-Anweisungen im Programmcode erfolgen.

verschiedenen Primary-Objekten durchgeführt. Der Mittelwert dieser Ergebnisse ist in folgenden Tabellen zu finden:

Serialisierungsdauer: in jeweils n Dateien

Anzahl Primary-Objekte und Dateien	Dauer [ms]
100	2660
500	15200
1000	34060

Deserialisierungsdauer: aus den n Dateien in eine `ArrayList`

Anzahl Dateien	Dauer [ms]
100	3100
500	12000
1000	24050

Speicherverbrauch des Serialisierung- und Deserialisierungsvorganges:

Anzahl Primary-Objekte und Dateien	Speicher [MB]
100	17
500	64
1000	126

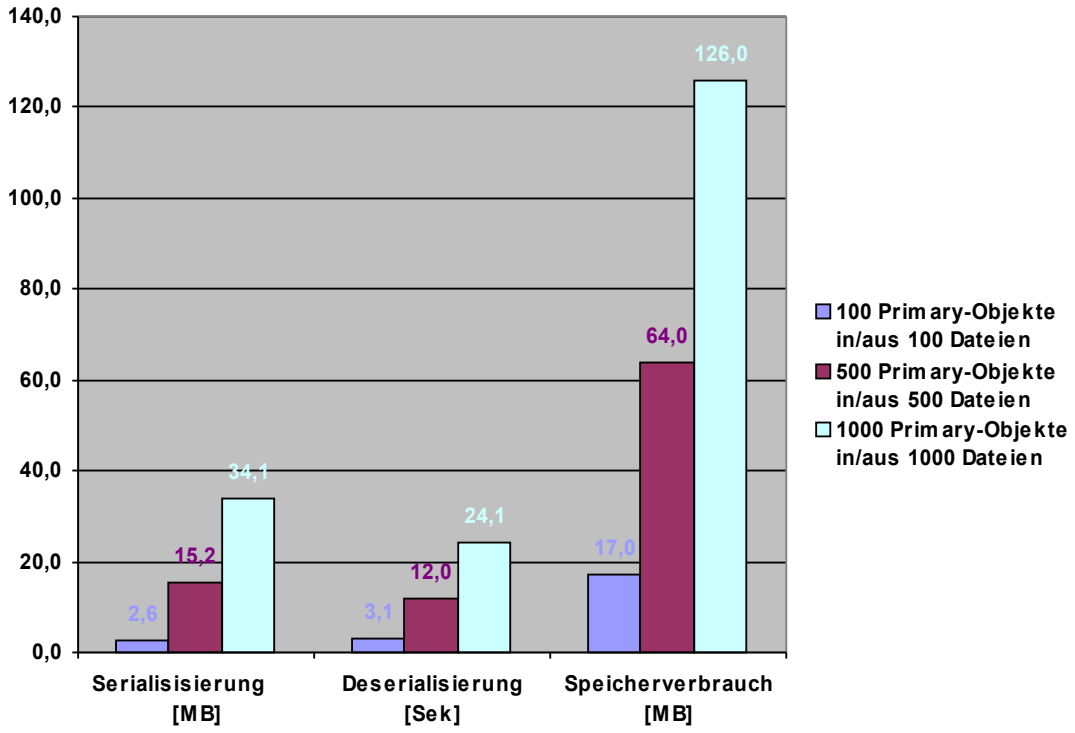


Abbildung 13„Diagramm Serialisierung, Deserialisierung und Speicherverbrauch“

Die verschiedenen Tests wurden zu Beginn mit der intelliBO Version 3.1 und dem MM-SQL-Datenbanktreiber 2.0.4 durchgeführt. Dieser Datenbanktreiber wurde von intelliBO als offiziell zu verwendender Datenbanktreiber für MySQL deklariert. Die Vorabversion 3.2 wurde auch mit diesem und interessehalber noch mit dem offiziellen JDBC-Treiber von MySQL Connector/J V3.0 durchgeführt. Durch Nutzung dieses Treibers konnten keine Unterschiede in den Datenbankeinträgen bezogen auf die verwendeten Datentypen des Prototyps festgestellt werden.

Im folgenden werden nun die gemessenen Zeiten der jeweiligen erfolgreichen (commit-Vorgang) und abgebrochenen[1] (rollback-Vorgang) Datenbank-Operationen und der dadurch verbrauchte Arbeitsspeicher der intelliBO V3.2[2] mit dem JDBC-Treiber MM-SQL-Treiber V2.0.4 und dem offiziellen MySQL- JDBC- Treiber Connector/J V3.0 aufgelistet:

4.1.3.1Insert-Operation

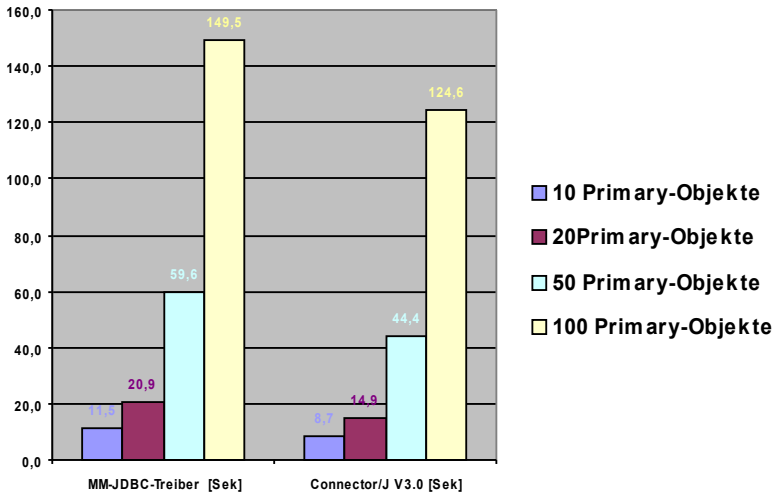
n Anzahl von `Primary`-Objekten aus einer `ArrayList` in die Datenbank eintragen:

MM-SQL-Treiber V2.0.4

Anzahl	[ms]
10	11495
20	20951
50	59636
100	149530

Connector/J V3.0:

Anzahl	[ms]
10	8713
20	14952
50	44357
100	124570



Rollback:

[A]: Zeit in [ms], in denen die Hauptobjekte instanziiert, in einer `ArrayList` zwischengespeichert und der Methode `makePersistentAll(List)` übergeben wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	2405	234
20	3465	990
50	6405	910
100	10310	4354

Fazit:

Es ist zu erkennen, dass der offizielle MySQL-JDBC-Treiber Connector/J V3.0 ca. 20-25 % schneller als der MM-JDBC-Treiber ist. Durch Aufruf der Methode `makePersistent(obj)` bzw. `makePersistentAll(list)` wechseln das Objekt bzw. alle sich in der List befindlichen Objekte vom Zustand transient in den Zustand persistent-new. Durch Aufruf der Methode `commit()` wird der komplette Objektgraph durchlaufen, und alle von diesen Objekten aus referenzierten Objekte (Eigenschaft: Persistenz durch Erreichbarkeit[3]) als persistent deklariert und in der zugrunde liegenden

[1] Es wurde bei den Tests mit rollback jeweils überprüft, dass keine Änderung persistent wurde

[2] Hier werden beabsichtigt nur Ergebnisse der Version 3.2 aufgelistet, da diese Version ca. 30-40% performanter als die Version 3.1 ist

[3] Der Begriff „Persistenz durch Erreichbarkeit“ wurde in Abschnitt 2.2.1.3 eingeführt

Datenbank gespeichert. Nun wechseln die abgespeicherten Objekte in den Zustand hollow. Der commit-Vorgang geht sehr auf Kosten der Performance und des Speicherverbrauchs, was mit dem Profiler OptimizeIt nachgewiesen wurde. Wird die aktuelle Transaktion durch Aufruf der Methode `rollback()` abgebrochen, so wechselt dass sich im Zustand persistent-new befindliche Objekt wieder in den Zustand transient.

4.1.3.2Delete-Operation

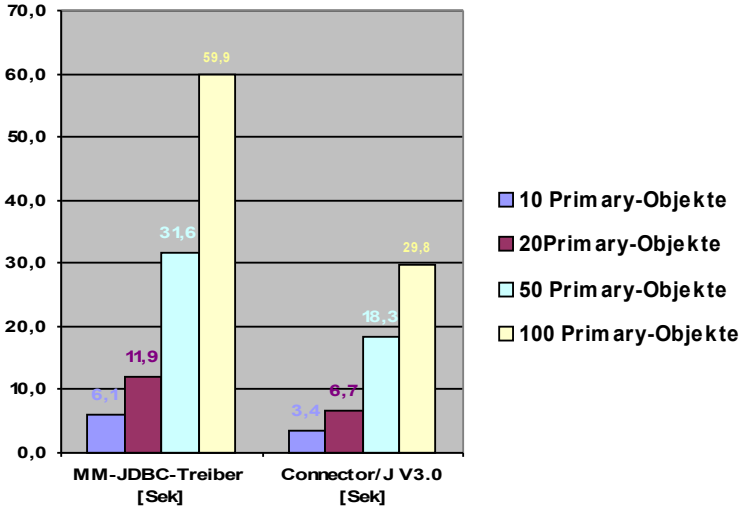
n `Primary`-Objekte, die sich im Speicher befinden werden innerhalb einer Transaktion in der Datenbank gelöscht:

MM-SQL-Treiber V2.0.4:

Anzahl	[ms]
10	6120
20	11960
50	31600
100	59860

Connector/J V3.0:

Anzahl	[ms]
10	3415
20	6710
50	18330
100	29874



Rollback:

[A]: Zeit in [ms], in der die sich im Speicher befindlichen Objekte der Methode `deletePersistentAll(List)` übergeben wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	1021	261
20	1061	431
50	2294	1011
100	6931	1892

Fazit:

Bei der `delete`-Operation ist der JDBC-Treiber Connector/J V3.0 ca. 45-50% schneller als der MM-JDBC-Treiber. Die aus der Datenbank gelesenen Objekte wechseln durch die Abfrage vom Zustand hollow in den Zustand persistent-clean. Durch Aufruf der Methode `deletePersistentAll` wird den Objekten der Zustand persistent-clean entzogen und sie wechseln in den Zustand persistent-deleted. Wird nun die aktuelle Transaktion erfolgreich durch Aufruf der Methode `commit()` beendet, so wechseln die sich in der `List` befindlichen Objekte in den Zustand transient. Wird hingegen die aktuelle Transaktion durch Aufruf der Methode `rollback()` abgebrochen, so wechseln die Objekte von dem Zustand persistent-deleted erneut in den Zustand hollow.

4.1.3.3Query-Operation

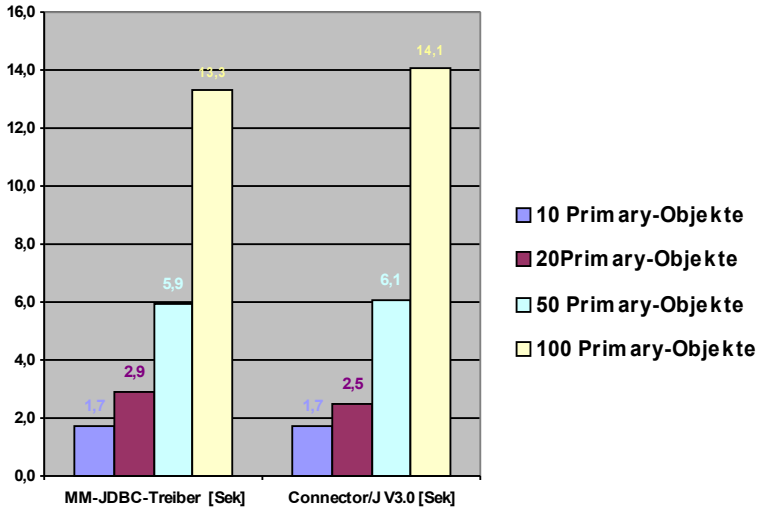
n Anzahl von `Primary`-Objekten werden aus der Datenbank gelesen und in einer `ArrayList` zwischengespeichert:

MM-SQL-Treiber V2.0.4:

Anzahl	[ms]
10	1743
20	2994
50	5898
100	13330

Connector/J V3.0:

Anzahl	[ms]
10	1773
20	2492
50	6050
100	14052



Rollback:

[A]: Zeit in [ms], in der eine Abfrage an die Datenbank erfolgte und alle Hauptobjekte in einer `ArrayList` gespeichert wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	471	360
20	641	441
50	1251	1162
100	5328	2454

Fazit:

Die `Query`-Operation wurde innerhalb einer aktiven Transaktion ausgeführt. Bei intelliBO ist es auch möglich, dass Query-Operationen ohne Transaktionen ausgeführt werden können. Dies muss in der Klasse `ConfJDO` explizit durch Setzen des Flag `nonTransactionalRead` auf `true` bei der Konfiguration der `PersistenceManagerFactory` erfolgen. Zwischen den beiden verwendeten JDBC-Treibern ist ein vernachlässigbarer Unterschied hinsichtlich der Performance der Query-Operation zu erkennen. Nach Ausführung der Query-Operation wechseln die ausgelesenen Objekte von dem Zustand hollow in den Zustand persistent-clean. Findet nun keine Änderung an den ausgelesenen Objekten statt, so wechselt der Zustand nach Aufruf der Methode `commit()` oder `rollback()` erneut in den Zustand hollow.

4.1.3.4Update-Operation

Allen Objekten der Klasse `SecondaryFirst` wurde ein neuer Bezeichner zugewiesen, d.h. die sich im Speicher befindlichen Objekte wurden verändert und somit mit der Datenbank synchonisiert.

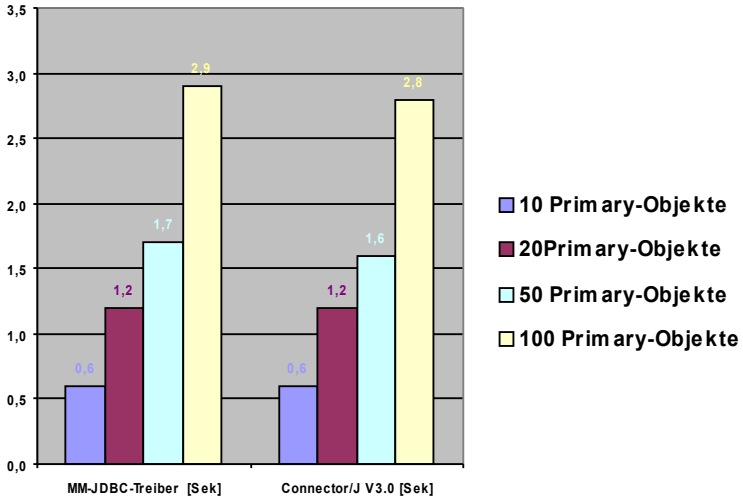
Die Anzahl bezieht sich auf die in der Datenbank gespeicherten `Primary`-Objekten. Die daraus resultierende Anzahl von `SecondaryFirst`-Objekten steht jeweils in Klammern.

MM-SQL-Treiber V2.0.4:

Anzahl	[ms]
10 (200)	660
20 (400)	1210
50 (1000)	1742
100 (2000)	2978

Connector/J V3.0:

Anzahl	[ms]
10 (200)	560
20 (400)	1160
50 (1000)	1582
100 (2000)	2794



Rollback:

[A]: Zeit in [ms], die die Änderung an den `SecondaryFirst`-Objekten benötigte

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	300	50
20	460	71
50	761	110
100	3876	220

Fazit:
Auch bei der `Update`-Operation ist zwischen den beiden verwendeten JDBC-Treibern ein vernachlässigbarer Unterschied zu erkennen. Die Objekte wechseln vom Zustand hollow in den Zustand persistent-clean, und verbleiben in diesem, solange keine Änderungen an den ausgelesenen Objekten durchgeführt werden. Durch Aufruf der Methode `setString(stringValue)` wird allen ausgelesenen `SecondaryFirst`-Objekten ein neuer Bezeichner zugewiesen. Dadurch wechseln die Objekte in den Zustand persistent-dirty. Nun liegt ein inkonsistenter Zustand zwischen den Werten in den Objekten und der Datenbank vor. Wird nun die aktuelle Transaktion durch Aufruf der Methode `commit()` erfolgreich beendet, so wechselt der Zustand der Objekte von persistent-dirty in den Zustand hollow, und die Änderungen werden in der Datenbank abgespeichert. Wird die aktuelle Transaktion durch Aufruf von `rollback()` abgebrochen, so findet der gleiche Zustandswechsel statt, aber die Änderungen werden verworfen, d.h. sie werden nicht in der Datenbank abgespeichert.

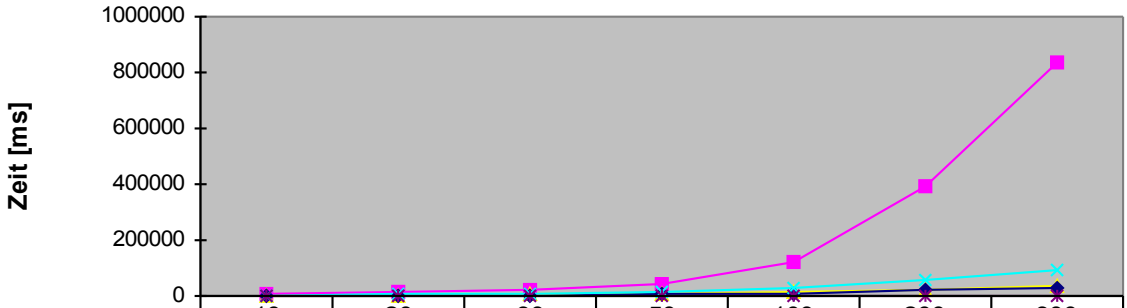
4.1.3.5Speicherverbrauch

Die Datenhaltung von 100 `Primary`-Objekten innerhalb einer `ArrayList` umfasst ca. 6MB im Arbeitsspeicher. Werden diese 100 `Primary`-Objekte innerhalb einer Transaktion mit Hilfe der JDO-Implementierung intelliBO in die Datenbank MySQL

eingetragen, so steigt der Arbeitsspeicher auf ca. 124MB an. Mit Hilfe des Profiler[1] Borland OptimizeIt [Borl03] wurde der Arbeitsspeicher und alle erzeugten Objekte betrachtet. Es war zu erkennen, dass sehr viele generischen bzw. Wrapper-Objekte durch intelliBO erzeugt werden. Durch einige Cache-Optimierungen[2] konnte dieses Speicherproblem nicht behoben werden. Aussage von Signsoft war, dass wenn sehr viele komplexe Objekte innerhalb einer Transaktion in die Datenbank eingetragen werden, es zu Speicherproblemen kommen kann, da intelliBO nicht Speicher-optimiert, sondern Performance-optimiert erstellt wurde.

4.1.3.6Performance und Ressourcenverbrauch intelliBO V3.2

Obige Datenbank-Operationen wurden auch mit der aktuell verabschiedeten intelliBO V3.2 durchgeführt. Es folgt eine Übersicht der erzielten Ergebnisse:



	10	20	30	50	100	200	300
insert	10075	16085	23065	44276	123735	396324	835892
query	1352	2413	3946	6691	12719	24026	37807
update	1102	2063	3205	4987	9745	19359	29514
delete	3275	6510	8903	14341	29053	59489	94702
MB	28	41	46	72	118	211	342

Anzahl Primary-Objekte

Die Grundeinstellung von intelliBO ist, dass jede an der Vererbung beteiligte Klasse in einer eigenen Tabelle abgebildet wird. Diese Vorgehensweise wurde übernommen. Erbt eine Klasse z.B. nur ein Attribut, ist es ratsam beide Klassen in einer Tabelle manuell abzubilden.

Die erbende Klasse erbt auch die Objekt-ID der Oberklasse, d.h. für diese Klasse ist bei Verwendung des eigenen Identitätenkonzept keine PrimaryKey-Klasse zu erstellen. Lediglich bei den CREATE-TABLE-Statements ist darauf zu achten, dass die an der Vererbung beteiligten Klassen bzw. die zugehörigen Tabellen jeweils den gleichen PrimaryKey besitzen.

Bsp: CREATE-TABLE-Statements von BasePrimary und Primary

CREATE TABLE TBASEPRIMARY (**FINTVALUEBP** INT(10) not null, **FCLASSTYPE** VARCHAR(50), FSTRINGVALUEBP VARCHAR(50), FBOOLVALUEBP TINYINT(3), PRIMARY KEY (FINTVALUEBP)) type=innoDB;

[1]

[2] War zwar nicht Bestandteil der Diplomarbeit, wurde aber interessehalber ausprobiert

```
CREATE TABLE TPRIMARY (FINTVALUEBP INT(10) not null, FBOOLVALUEP TINYINT(3), FSTRINGVALUEP VARCHAR(50),
FPROPERTIESP BLOB, PRIMARY KEY (FINTVALUEBP)) type=innoDB;
```

In der Spalte `FCLASSTYPE` der Tabelle `TBASEPRIMARY` wird der Klassenname der von dieser erbenden Klasse angegeben. Dies bringt den Vorteil, dass bei Abfragen der Typ des Objektes erhalten bleibt.

Auszug aus dem Persistenz-Deskriptor der Oberklasse `BasePrimary`:

```
<extension key="inheritance" vendor-name="ssibo">
<extension key="provider" value="classname" vendor-name="ssibo"/>
<extension key="field-name" value="FCLASSTYPE" vendor-name="ssibo"/>
<extension key="subclass" value="prototype.model.Primary" vendor-name="ssibo"/>
</extension>
```

Durch den Schlüsselwert „`inheritance`“ wird die Vererbung eingeleitet. Die von der Oberklasse erbenden Unterklassen müssen nicht extra angegeben werden. Falls dies doch geschieht kann intelliBO optimaler arbeiten.

Auszug aus dem Persistenz-Deskriptor der von der Klasse `BasePrimary` erbenden Klasse `Primary`:

```
<class identity-type="application" name="Primary" persistence-capable-superclass="BasePrimary" >
```

Durch Angabe des Schlüssels `<persistence-capable-superclass>` des Elementes `<class>` wird dieser Klasse die zugehörige Oberklasse bekannt gemacht.

4.1.5Assoziationen

Bei der Implementierung wurden 1...1- und 1...N- Assoziationen verwendet, auf die dadurch aufgetretenen Probleme wird in diesem Abschnitt eingegangen. Für die Abbildung der 1...N-Assoziationen wurden die Datentypen `ArrayList` und `HashMap` verwendet. Die jeweiligen Assoziationen werden im Persistenz-Deskriptor bei Verwendung von intelliBO wie folgt beschrieben:

```
<extension key="mapping" value="XXXX" vendor-name="ssibo"/>
```

Die jeweils verwendete Abbildungsregel wird durch Ersetzen der XXXX durch entweder:

- direct oder
- one-to-one oder
- one-to-many oder
- serial oder
- code oder
- many-to-many

eingeleitet. Verwendet wurden die ersten vier. Eine Erklärung anhand eines Beispiels über direct, one-to-many und serial erfolgte schon in Abschnitt 3.2.1.2, daher wird an dieser Stelle kurz die 1...1-Assoziation erläutert:

One-to-one:

```
<field name="primaryFirst">
    <extension key="jdbc" vendor-name="ssibo">
    <extension key="mapping" value="one-to-one" vendor-name="ssibo">
    <extension key="element-type" value="prototype.model.PrimaryFirst" vendor-name="ssibo"/>
    </extension>
    </extension>
```

```
</field>
```

One-to-one bedeutet, dass ein Objekt eine Referenz auf ein anderes Objekt hält. Oben ist ein Auszug aus dem Persistenz-Deskriptor der Klasse `PrimarySecond` aufgelistet. Ein Objekt dieser Klasse hält eine Referenz auf ein Objekt der Klasse `PrimaryFirst`.

Probleme beim Java-Datentyp `Properties`:
Wird der Java-Datentyp `Properties` verwendet, so wird dieser unter Zuhilfenahme des Java-Serialisierungsmechanismus in der Datenbank als SQL-Datentyp `BLOB` gespeichert. Das daraus resultierende Problem ist, dass nicht gezielt nach einem Wert anhand eines Schlüssels des `Properties` gesucht werden kann. Es muss zuerst der gesamte `BLOB` aus der Datenbank gelesen werden, dann kann dieser nach dem Schlüssel durchforstet werden. Dies geht auf Kosten der Performance, da erst der gesamte `BLOB` deserialisiert werden muss. Es besteht nicht die Möglichkeit, dass der Java-Datentyp `Properties` auf eine eigene Tabelle[1] abgebildet werden kann.

Probleme bei den Java-Datentypen `Map` und `List`:
Werden in einem Objekt Attribute vom Java-Datentyp `Map` und `List` verwendet, die beide als Sammlung von Objekten desselben Datentyp dienen, trat ein Problem bei der Verwendung der Standardeinstellung von intelliBO auf. Dies wird an nachfolgendem Beispiel erläutert:

Objekte der Klasse `Primary` besitzen eine `List` und eine `Map`, in denen jeweils `Secondary`-Objekte verwaltet werden. Wird die Standardeinstellung von intelliBO verwendet, so wird in der zur Klasse `Secondary` gehörenden Tabelle `TSECONDARY` eine Spalte namens `FTPRIMARY_FINTVALUEBP` definiert. Diese Spalte stellt den Fremdschlüssel auf die Tabelle `TPRIMARY` dar. Werden nun durch intelliBO die instanziierten Hauptobjekte der Klasse `Primary` in die Tabellen der Datenbank eingetragen, so treten keine Probleme auf, d.h. es wird alles ordnungsgemäß abgespeichert. Probleme treten auf, wenn auf die in der Datenbank gespeicherten `Primary`-Objekte zugegriffen wird, d.h. durch eine Abfrage werden alle `Primary`-Objekte aus der Datenbank geladen und in einer `List` zwischengespeichert. Wird nun auf die jeweilige `List` oder `Map` der Primary-Objekte zugegriffen und diese ausgegeben, so beinhalten diese nicht die ordnungsgemäß jeweils zugehörenden `Secondary`-Objekte. Dieses Problem wird gelöst, indem eine zusätzliche Spalte `FTPRIMARY_ARRAY` in der Tabelle `TSECONDARY` aufgenommen wird und im Persistenz-Deskriptor `primary.jdo` durch das Element `<reference>` eine zusätzliche Fremdschlüssel-Beziehung definiert wird (siehe Abschnitt 3.2.1.2).

4.1.6Komposition

Bei den Abbildungsregeln one-to-one, one-to-many und many-to-many kann jeweils das kaskadierende Löschen aktiviert werden. D.h. wenn das Objekt, dass Referenzen auf andere Objekte hält, gelöscht wird, so werden auch alle von diesem Objekt aus referenzierten Objekte gelöscht. Dies wird im jeweiligen Persistenz-Deskriptor durch das Element `<cascade-delete>` bei der Beschreibung des Attributes aktiviert.

```
<extension key="cascade-delete" value="true" vendor-name="ssibo"/>
```

[1] Als Spalten besäße diese Tabelle dann einen Fremdschlüssel auf die diesen Datentyp besitzende Klasse, jeweils eine Spalte für den Schlüssel und Wert

4.1.7 Graphische Entwicklungsumgebung

Die graphische Entwicklungsumgebung von intelliBO ist übersichtlich aufgebaut, selbsterklärend und leicht zu nutzen. Die zu speichernden Klassen werden eingelesen und es kann automatisch ein Persistenz-Deskriptor erstellt werden. Des weiteren wird ein Quelltext-Editor für diese bereitgestellt, so dass der Entwickler auch „von Hand" Änderungen vornehmen kann. Aus der Entwicklungsumgebung heraus können Werkzeuge zur Tabellenskript-Generation, Syntax-Überprüfung der erstellten Persistenz-Deskriptoren, Referentielle-Integrität-Überprüfung[1] und der Enhancement-Vorgang gestartet werden. Für die Bedienung der graphischen Entwicklungsumgebung wird auf das Handbuch von intelliBO verwiesen. [Sign02]

4.1.8 Abfragesprache(n)

Bei Verwendung von intelliBO kann auch eine andere Abfragesprache als JDOQL gewählt werden. Dies wird am `PersistenceManager` im Parameter `language` der Methode `newQuery(String language, Object obj)` angegeben. Folgende Angaben werden unterstützt:

- JDOQL Abfragesprache der JDO-Spezifikation
- RESULT_SET[2]
- SQL SQL als Abfragesprache

Des weiteren wird die gleichzeitige Verwendung von JDOQL und SQL unterstützt.

4.1.9 Unterstützte Systemumgebung

IntelliBO ist in der eigenständigen und eingebetteten Umgebung lauffähig (siehe Abschnitt 2.2.1.1).

4.2 Hibernate

In diesem Abschnitt werden die Ergebnisse der Evaluierung von Hibernate vorgestellt und auf Probleme, die während der Implementierung auftraten eingegangen. Verwendet wurde die zu Beginn der Evaluierung zur Verfügung stehende Version 1.2.3.

[1] Anhand dieser Überprüfung werden Abhängigkeiten zwischen den Tabellen erkannt und eine XML-Datei mit der Reihenfolge der UPDATE/INSERT/DELETE-Befehle erstellt. Diese kann von intelliBO genutzt werden, um die Statement-Reihenfolge zu ordnen

[2] Eine ResultSet-Abfrage kann verwendet werden, wenn die Ergebnismenge einer SQL-Anfrage nicht durch eine Sammlung von Objekten repräsentiert werden kann. Als Ergebnis wird eine Menge von Maps zurückgegeben, welche als Schlüssel den Namen der Tabellenspalte und als Wert den aktuellen Wert des Feldes enthalten [Sign02, S.118]

4.2.1 Automatische Generierung des Persistenz-Deskriptors

Es wurde das bei Hibernate mitgelieferte Kommandozeilen-orientierte Werkzeug `MapGenerator` zur automatischen Generierung der Persistenz-Deskriptoren getestet. Der Aufruf geschieht pro Klasse, d.h. es können nicht alle Persistenz-Deskriptoren auf einmal erstellt werden. Der Aufruf geschieht wie folgt:

java cirrus.hibernate.tools.MapGenerator –setUID="Primärschlüssel" –output="XML-Datei-Name" „Klassenname"

Die Ergebnisse der automatischen Generierung werden vorgestellt und dienen lediglich als ein Grundgerüst, an welchem noch manuelle Korrekturen durchgeführt werden müssen:

- Der Verweis zur DTD im Persistenz-Deskriptor wurde wie folgt http://hibernate.sourceforge.net/hibernate-mapping.dtd erstellt und muss auf http://hibernate.sourceforge.net/hibernate-mapping-1.1.dtd korrigiert werden

- Die an der Vererbung beteiligten Unterklassen wurden nicht auf eine eigene Tabelle abgebildet, sondern deren Attribute in die Tabelle der zugehörenden Oberklasse aufgenommen

- Für die Abbildung der Datentypen `List` und `Map` wird immer eine eigene Tabelle erstellt. Z.B. werden in der zur `Map` zugehörigen Tabelle die Fremdschlüssel und der Schlüssel der `Map` gehalten

- Bei einer 1...1-Assoziation wird das referenzierte Objekt immer als ein eingebettetes Objekt betrachtet, d.h. die Attribute dieses Objektes werden in die Tabelle des es besitzenden Objektes aufgenommen

4.2.2 Automatisches Erstellen der zugehörigen Tabellen

Zur automatischen Erstellung der Tabellen wurde das mitgelieferte Kommandozeilen-basierte Werkzeug `SchemaExport` getestet. Dieses Werkzeug erzeugt auf Basis des zuvor erstellen Persistenz-Deskriptors ein SQL-Skript passend zur verwendeten Datenbank. Das Werkzeug liest die Konfigurationsdatei „`hibernate.properties`" aus und erhält somit Informationen über die verwendete Datenbank. Der Aufruf geschieht wie folgt:

java cirrus.hibernate.tools.SchemaExport –output="Skript-Dateiname" „Persistenz-Deskriptor"

Durch Angabe von zusätzlichen Parametern ist es möglich, dass dieses Skript auch direkt auf der Datenbank ausgeführt wird. (siehe Dokumentation [Hibe03]).

Es werden für alle Tabellen drop- und create-Anweisungen, des weiteren alle Fremdschlüssel erzeugt.

In dem erstellten Skript wurden keine gravierenden Fehler festgestellt. Lediglich das ein Statement abschliessende Semikolon muss manuell eingefügt werden, sowie die Nutzung des Tabellentyps innoDB „`type=innoDB`" in die Create-Anweisungen aufgenommen werden. Dieses erzeugte Skript wurde bis auf obige Änderungen, sowie Einschränkungen der Wertebereiche der verwendeten SQL-Datentypen übernommen und ist im Anhang 6.7 zu finden.

4.2.3Integrationsaufwand für bereits existierende Klassen

Gegenüber intelliBO verändert Hibernate nicht den Bytecode der kompilierten Klassen, sondern nutzt die Java Reflection-API, um zur Laufzeit Informationen über die Daten der Objekte zu erhalten um somit dynamisch SQL-Code erzeugen zu können. Jede Java-Klasse kann durch Hibernate in die verwendete Datenbank abgespeichert werden. Lediglich muss sie einen Default-Konstruktor und get/set- Methoden (JavaBeans-konform[1]) besitzen. Bei Datentypen muss die implementierte Schnittstelle und nicht die konkrete Klasse angegeben werden, d.h. bei Verwendung einer `ArrayList` als Datentyp wird nicht diese, sondern bei der Deklaration `List` angegeben. Dies ist notwendig, da Hibernate Instanzen von `Map`, `List` und `Set` durch eigene Implementierungen dieser Schnittstellen ersetzt.

4.2.4Performance und Ressourcenverbrauch

Es wurden unter anderem die gleichen Tests wie auch bei intelliBO durchgeführt. Die Tests wurden jeweils mit der Verwendung der „trägen Initialisierung“ und ohne „träge Initialisierung“ ausgeführt. Es folgt eine Auflistung der Ergebnisse der einzelnen Operationen.

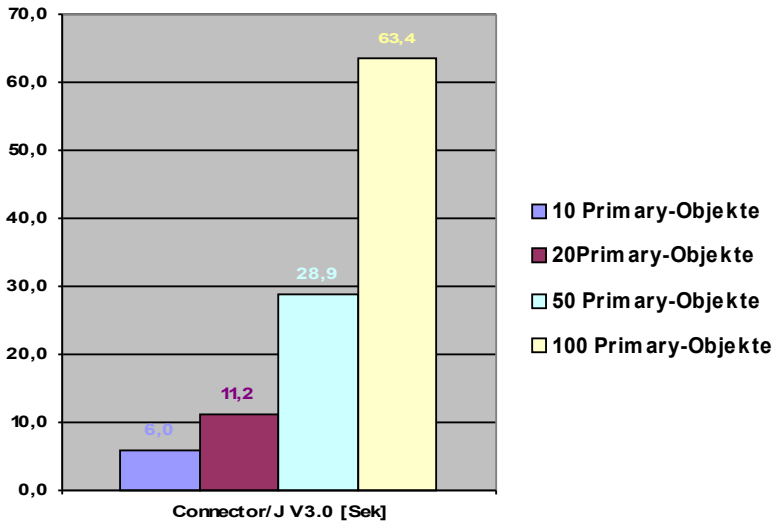
4.2.4.1Insert-Operation

Es wurden keine Abweichungen bei Verwendung mit und ohne der „`lazy`“ Initialisierung festgestellt.

n Primary-Objekte werden aus einer `ArrayList` in die Datenbank eintragen:

Connector/J V3.0:

Anzahl	[ms]
10	6000
20	11200
50	28900
100	63400



Rollback:

[A]: Zeit in [ms], in denen die Hauptobjekte instanziiert, die referenzierten Objekte der Methode `save(obj)` übergeben und in einer `ArrayList` zwischengespeichert wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]

[1] z.B.: bei dem Attribut intValueBP: getIntValueBP() und setIntValueBP(int intValueBP); ansonsten kommt es zu einer NullPointerException

10	1021	0
20	1422	0
50	3115	10
100	5368	20

Fazit:

Durch Aufruf der Methode `save(obj)` der Schnittstelle `Session` wechseln die sich im Speicher befindlichen Objekte vom Zustand transient in den Zustand persistent. Die Methode `commit()` der Schnittstelle `Transaction` veranlasst eine Speicherung in der Datenbank. Es ist eine lineare Laufzeit zu erkennen.

4.2.4.2Delete-Operation

Die n sich im Speicher befindlichen `Primary`-Objekte wurden innerhalb einer Transaktion gelöscht. Es werden die Zeiten mit und ohne Verwendung der „lazy“ Initialisierung aufgelistet.

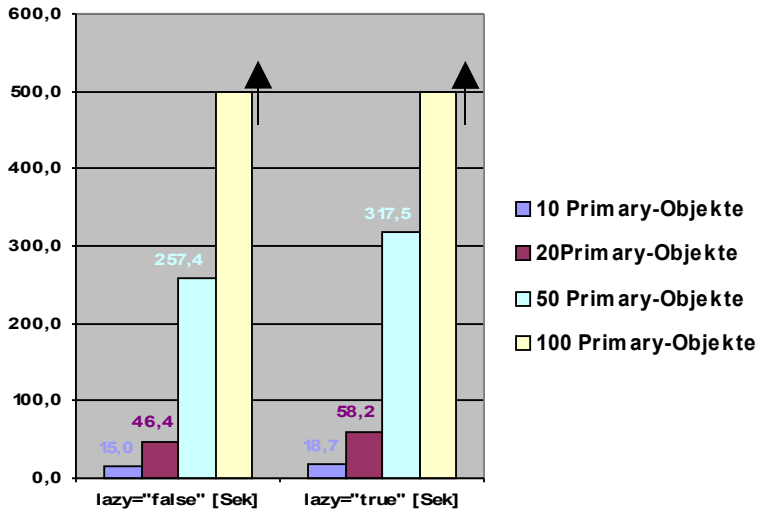
n `Primary`-Objekten werden in der Datenbank gelöscht:

lazy=“false“:

Anzahl	[ms]
10	15000
20	46399
50	257395
100	1454512

lazy=”true”:

Anzahl	[ms]
10	18700
20	58200
50	317466
100	



Rollback:

[A]: Zeit in [ms], in der die sich im Speicher befindlichen `Primary`-Objekte der Methode `delete(obj)` übergeben wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	1021	261
20	1061	431
50	2294	1011
100	6931	1892

Fazit:
Den sich im Speicher befindlichen Objekten, welche den Zustand persistent besitzen, wird durch Aufruf der Methode `delete()` der Schnittstelle `Session` der Zustand persistent entzogen. Sie wechseln in den Zustand transient über. Es ist eine exponentielle Laufzeit zu erkennen. Nach Rüchsprache mit dem Entwickler von Hibernate wurde bestätigt, dass wenn sehr viele Objekte innerhalb einer Transaktion gelöscht werden, dies eine hohe Laufzeit beansprucht. Falls die Performance im Vordergrund steht, sollen für diesen Fall direkte SQL-Statements ausgeführt werden.

4.2.4.3Query-Operation

Es werden die Zeiten bei Verwedung der „lazy“ Initialisierung und ohne aufgelistet.

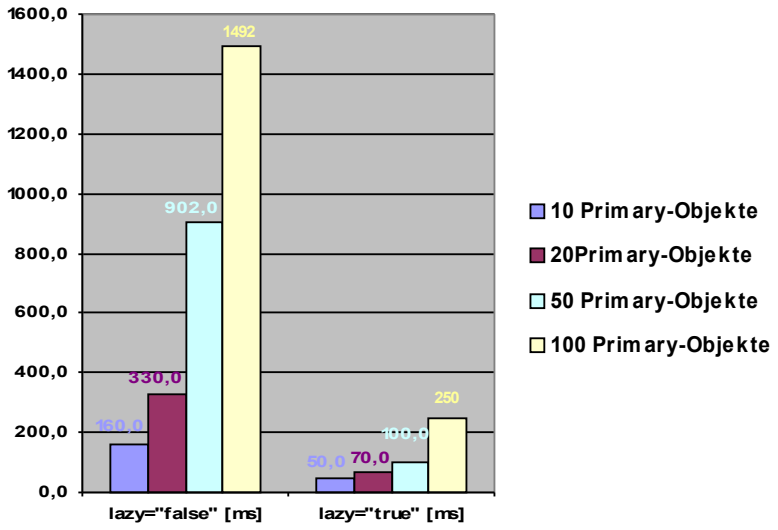
n `Primary`-Objekte werden aus der Datenbank gelesen und in einer `ArrayList` zwischengespeichert:

lazy=“false“:

Anzahl	[ms]
10	160
20	330
50	902
100	1493

lazy=”true”:

Anzahl	[ms]
10	50
20	70
50	100
100	250



Rollback:

[A]: Zeit in [ms], in der eine Abfrage an die Datenbank erfolgte und alle Hauptobjekte in einer `ArrayList` gespeichert wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	0	0
20	0	10
50	0	10
100	0	20

Fazit:

Anhand der Objekt-Id der gespeicherten Objekte, werden diese durch Aufruf der Methode `load(Class name, Serializable id)` aus der Datenbank innerhalb einer Transaktion ausgelesen. Die ausgelesenen Objekte besitzen den Zustand persistent.

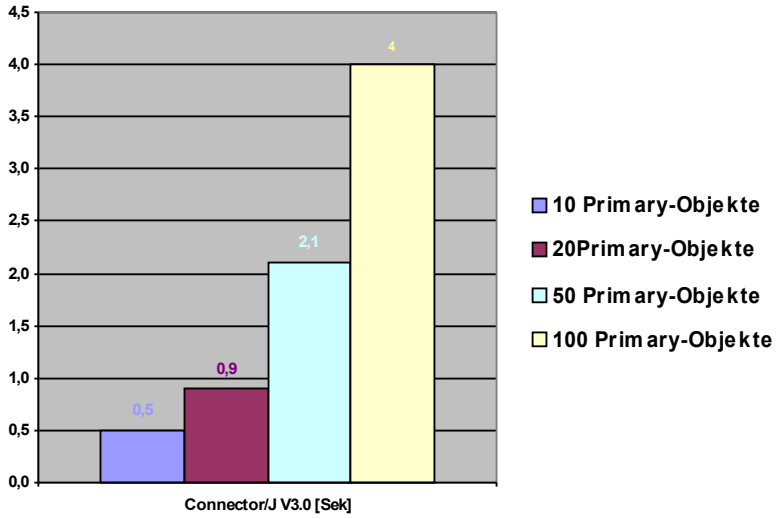
4.2.4.4Update-Operation

Allen Objekten der Klasse `SecondaryFirst` wurde ein neuer Bezeichner zugewiesen, d.h. die sich im Speicher befindlichen Objekte wurden verändert und somit mit der Datenbank synchronisiert:

Es wurden keine Unterschiede zwischen Verwendung mit und ohne der „lazy“ Initialisierung festgestellt. Die Anzahl bezieht sich auf die `Primary`-Objekte. Die daraus resultierende Anzahl von `SecondaryFirst`-Objekten steht jeweils in Klammern.

Connector/J V3.0:

Anzahl	[ms]
10 (200)	461
20 (400)	941
50 (1000)	2143
100 (2000)	3995



Rollback:

[A]: Zeit in [ms], in der die Änderung durchgeführt wurde

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	0	10
20	10	10
50	10	20
100	10	31

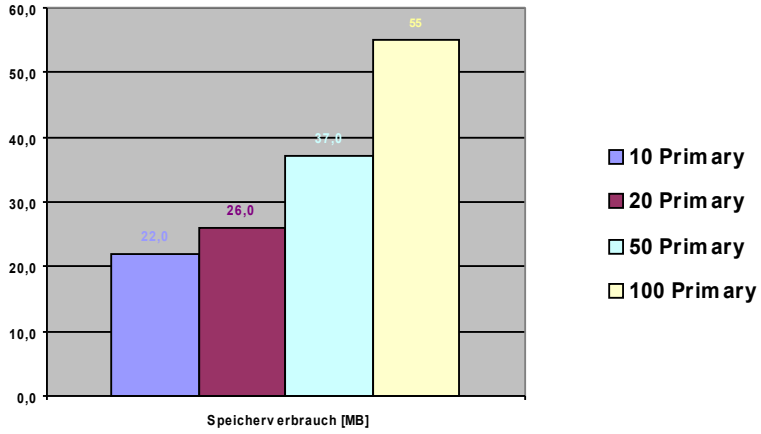
Fazit:

Durch Aufruf der Methode `setString(stringValue)` wurde allen im Speicher befindlichen `SecondaryFirst`-Objekten ein neuer Bezeichner zugewiesen. Dadurch liegt ein inkonsistenter Zustand zwischen den Werten in den Objekten und der Datenbank vor. Wird nun die Transaktion durch Aufruf der Methode `commit()` der Schnittstelle `Transaction` beendet, so werden die geänderten Werte der Objekte im Speicher mit der Datenbank synchronisiert. Es ist eine lineare Laufzeit zu erkennen.

4.2.4.5Speicherverbrauch

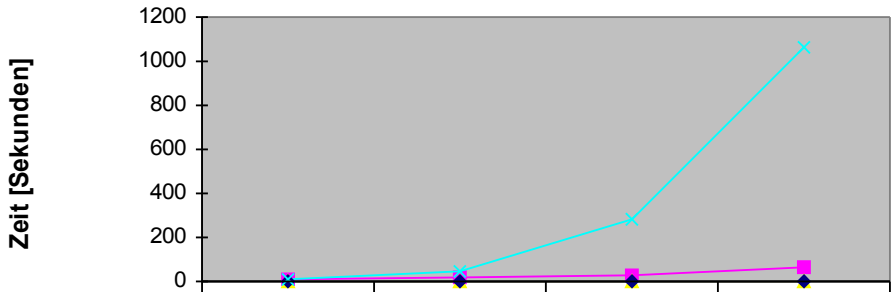
Es folgt eine Auflistung des Speicherverbrauchs, den der Testfall (obige Operationen) benötigt. Die Angaben wurden im Taskmanager abgelesen:

Anzahl	[MB]
10	22
20	26
50	37
100	55



4.2.4.6Performance unter Verwendung der Datenbank HSQL

Obige Datenbank-Operationen wurden auch mit der Datenbank HSQL jeweils im embedded- und Server-Modus durchgeführt:



	10	20	50	100
insert	6,3	13,7	31,1	67,4
query	0,4	0,7	1,5	3,7
update	0,5	1,1	2,2	4,3
delete	13,1	48,1	281	1061

Abbildung 14„Performance der DB-Operationen unter HSQL im Server-Modus“

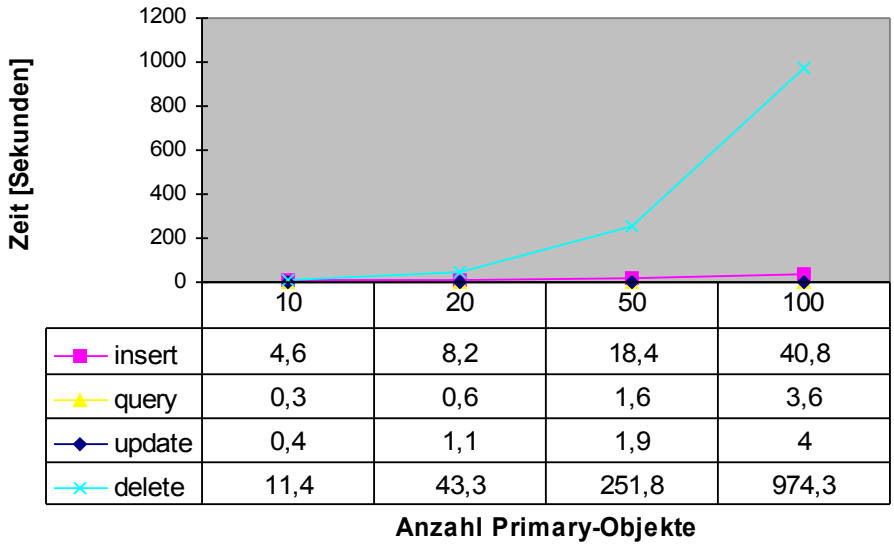


Abbildung 15„Performance der DB-Operationen unter HSQL im embedded-Modus“

4.2.5Vererbung

Bei Hibernate gibt es die Möglichkeit, dass die an der Vererbung beteiligten Klassen mit in der Tabelle der Oberklasse oder aber dass die Unterklassen jeweils in einer eigenen Tabelle abgebildet werden. Die Unterklassen erben die ObjektID der Oberklassen und die Beziehung zwischen den Tabellen wird über Fremdschlüssel hergestellt.

4.2.6Assoziationen

Hibernate unterstützt alle gängigen Assoziationen. Die verwendeten Assoziationen wurden schon im Abschnitt 3.2.2.3 vorgestellt, so dass hier nur die Probleme, die während der Verwendung auftraten, erläutert werden.

Werden in einem Objekt mehrere Datentypen, z.B. `List` und `Map`, verwendet, die die gleichen Elementtypen verwalten, so muss jeweils eine eigene Spalte für den Fremdschlüssel definiert werden.

4.2.7 Komposition

Auch Hibernate unterstützt das kaskadierende Löschen und es wird im Persistenz-Deskriptor bei den Elementen: `<many-to-one>`, `<one-to-one>` und `<many-to-many>` durch das Attribut `cascade="delete"` aktiviert.

4.2.8 Graphische Entwicklungsumgebung

Hibernate liegt keine graphische Entwicklungsumgebung bei, sondern Kommandozeilen-basierte Hilfswerkzeuge, auf die in den vorherigen Abschnitten schon eingegangen wurde.

4.2.9 Abfragesprache(n)

Hibernate definiert eine eigene Abfragesprache, um auf Objekte in der Datenbank zuzugreifen. Sie stellt eine Mischung zwischen SQL- und einer objektorientierten Syntax dar.

4.2.10 Unterstützte Systemumgebung

Hibernate ist in der eigenständigen und in der eingebetteten Umgebung lauffähig (siehe Abschnitt 2.2.1.1).

4.3 Castor JDO

4.3.1 Integrationsaufwand für bereits existierende Klassen

Wie schon in Abschnitt 3.2.3 „Implementierung Castor JDO“ erwähnt, mussten einige Änderungen an den vorhandenen Klassen vorgenommen werden. Daher wird der interessierte Leser auf diesen Abschnitt mit Unterabschnitt „Durchgeführte Änderungen am Prototyp“ verwiesen.

Zur dynamischen SQL-Code-Erzeugung nutzt Castor JDO während der Laufzeit die Java Reflection-API um Informationen über die Daten der Objekte zu erhalten.

Bei der Erstellung der zu speichernden Klassen ist daruaf zu achten, dass diese einen Default-Konstruktor und get/set- Methoden nach der JavaBeans-Stilform besitzen.

4.3.2 Hilfswerkzeuge

Castor JDO liegen keine graphische Entwicklungsumgebung sowie keine Kommandozeilen-basierten Hilfswerkzeuge bei.

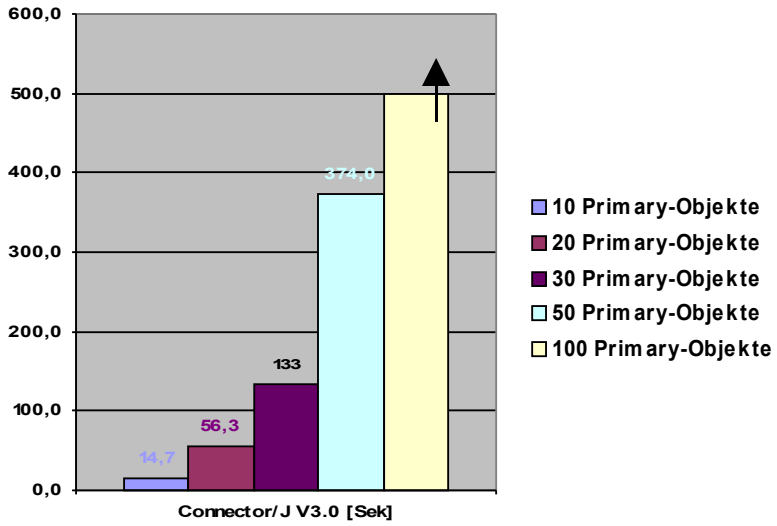
4.3.3Performance und Ressourcenverbrauch

4.3.3.1Insert-Operation

n `Primary`-Objekte werden aus einer `ArrayList` in die Datenbank MySQL eintragen:

Connector/J V3.0:

Anzahl	[ms]
10	14730
20	56265
30	132950
50	373900
100	1469371



Rollback:

[A]: Zeit in [ms], in denen die Hauptobjekte instanziiert, die referenzierten Objekte der Methode `create(obj)` der Schnittstelle `Database` übergeben und in einer `ArrayList` zwischengespeichert wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	10336	2807
20	35153	4462
30	80951	7046

Fazit:

Die erzeugten Objekte wechseln durch Aufruf der Methode `create(obj)` der Schnittstelle `Database` innerhalb einer Transaktion vom Zustand transient in den

Zustand persistent. Wird die aktuelle Transaktion durch die Methode `commit()` beendet, so wechseln die dadurch in die Datenbank eingetragenen Objekte erneut in den Zustand transient. Wird die Anzahl der zu speichernden Objekte verdoppelt, so ist eine Verdreifachung der Laufzeit festzustellen.

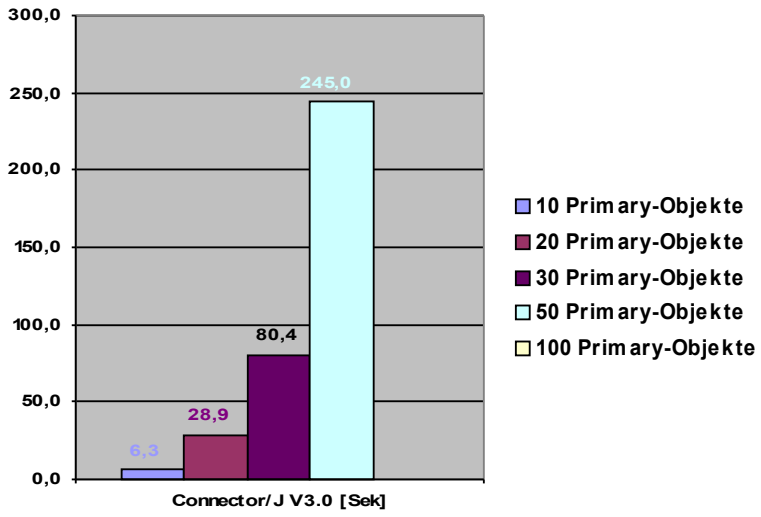
4.3.3.2Delete-Operation

Die n sich im Speicher befindlichen `Primary`-Objekte wurden innerhalb einer Transaktion gelöscht.

n `Primary`-Objekten werden in der Datenbank gelöscht:

Connector/J V3.0:

Anzahl	[ms]
10	6250
20	28944
30	80420
50	244940
100	998741



Rollback:

[A]: Zeit in [ms], in der die sich im Speicher befindlichen `Primary`-Objekte der Methode `remove(obj)` übergeben wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	5088	761
20	25593	2211
30	70213	8635

Fazit:

Da die sich im Speicher befindlichen Objekte den Zustand transient besitzen, muss erst durch Aufruf der Methode `update(obj)` der Schnittstelle `Database` diesen Objekten der Zustand persistent zugeteilt werden, so dass diese durch die Methode

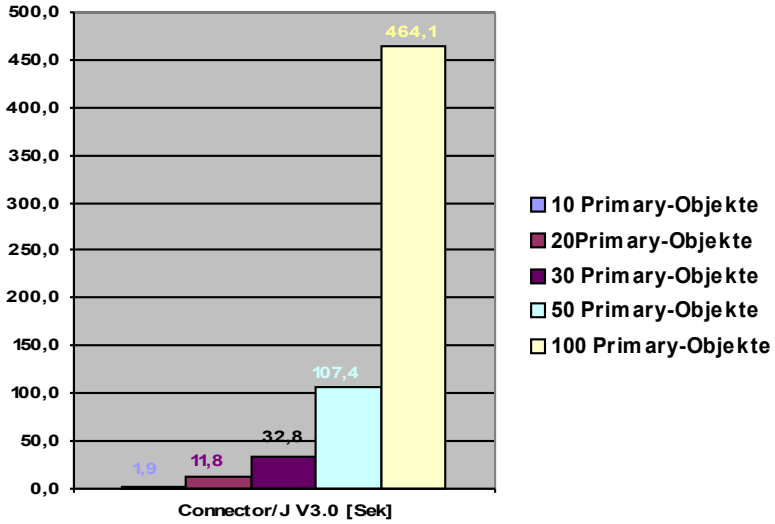
`remove(obj)` in der Datenbank gelöscht werden können. Es ist eine exponentielle Laufzeit zu erkennen.

4.3.3.3Query-Operation

n `Primary`-Objekte werden aus der Datenbank gelesen und in einer `ArrayList` zwischengespeichert:

Connector/J V3.0:

Anzahl	[ms]
10	1890
20	11810
30	32750
50	107441
100	464145



Rollback:

[A]: Zeit in [ms], in der eine Abfrage an die Datenbank erfolgte und alle Hauptobjekte in einer `ArrayList` gespeichert wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	1563	566
20	8723	1082
30	21540	3363

Fazit:

Obige Zeiten beziehen sich darauf, dass alle aus der Datenbank ausgelesenen `Primary`-Objekte vollständig initialisiert im Speicher vorliegen. Die ausgelesenen Objekte haben den Zustand persistent. Es ist eine exponentielle Laufzeit zu erkennen.

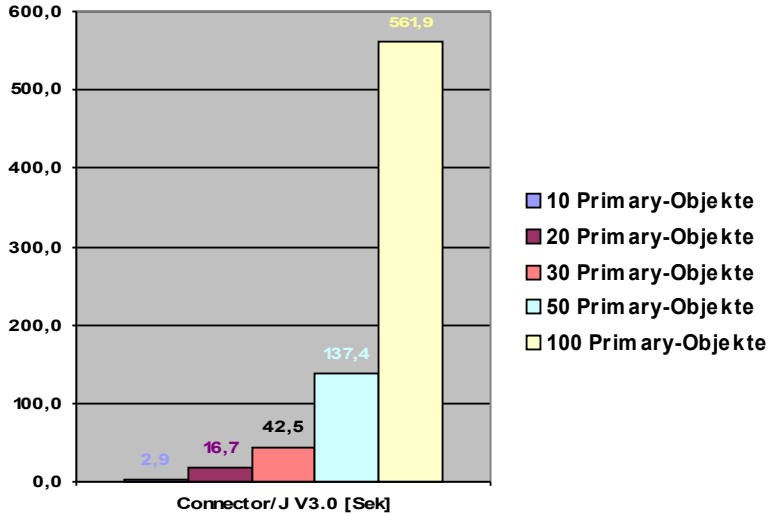
4.3.3.4Update-Operation

Allen Objekten der Klasse `SecondaryFirst` wurde ein neuer Bezeichner zugewiesen, d.h. die sich im Speicher befindlichen Objekte wurden verändert und somit mit der Datenbank synchronisiert:

Die Anzahl bezieht sich auf die `Primary`-Objekte. Die daraus resultierende Anzahl von `SecondaryFirst`-Objekten steht jeweils in Klammern.

Connector/J V3.0:

Anzahl	[ms]
10 (200)	2875
20 (400)	16685
30 (600)	42535
50 (1000)	137436
100 (2000)	561990



Rollback:

[A]: Zeit in [ms], in der die Änderung durchgeführt wurde

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	2043	2156
20	10556	4012
30	29460	8837

Fazit:

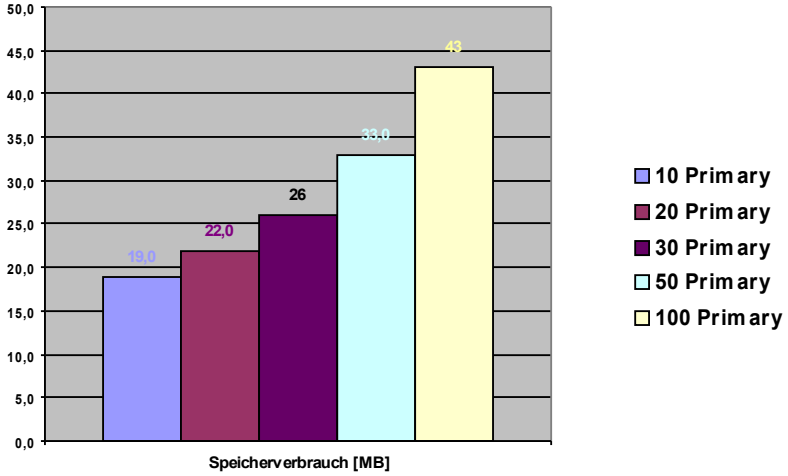
Durch Aufruf der Methode `update(obj)` der Schnittstelle `Database` wechseln die sich im Speicher befindlichen Objekte vom Zustand transient in den Zustand persistent. Durch Aufruf der Methode `setString(stringValue)` wurde allen im Speicher befindlichen `SecondaryFirst`-Objekten ein neuer Bezeichner zugewiesen. Dadurch liegt ein inkonsistenter Zustand zwischen den Werten in den Objekten und der Datenbank vor. Wird nun die Transaktion durch Aufruf der Methode `commit()`

beendet, so werden die geänderten Werte der Objekte im Speicher mit der Datenbank synchronisiert. Es ist eine exponentielle Laufzeit zu erkennen.

4.3.3.5Speicherverbrauch

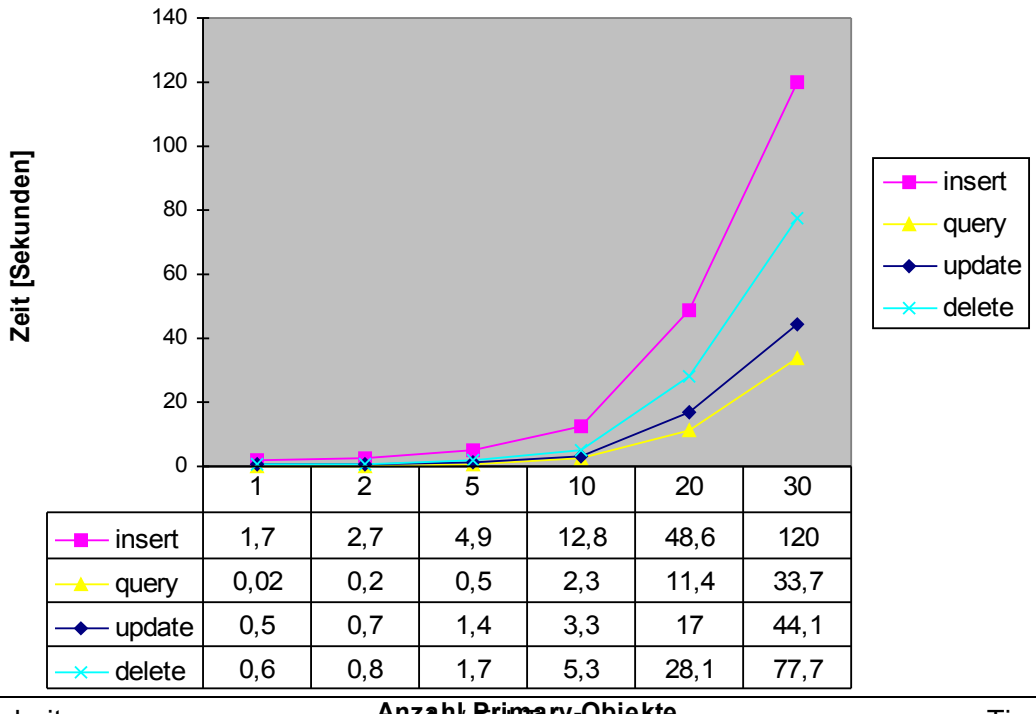
Es folgt eine Auflistung des Speicherverbrauchs, den der Testfall (obige Operationen) bei Verwendung der Datenbank MySQL benötigt. Die Angaben wurden im Taskmanager abgelesen:

Anzahl	[MB]
10	19
20	22
30	26
50	33
100	43



4.3.3.6Performance unter Verwendung der Datenbank HSQL

Es folgt eine Übersicht der durchgeführten Datenbank-Operationen unter Verwendung der Datenbank HSQL im eingebetteten Modus:



	1	2	5	10	20	30
insert	1,7	2,7	4,9	12,8	48,6	120
query	0,02	0,2	0,5	2,3	11,4	33,7
update	0,5	0,7	1,4	3,3	17	44,1
delete	0,6	0,8	1,7	5,3	28,1	77,7

Abbildung 16„Performance-Ergebnisse unter Verwendung der Datenbank HSQL im eingebetteten Modus“

4.3.3.7Vererbung

Wie auch bei Hibernate gibt es die Möglichkeit, dass die an der Vererbung beteiligten Klassen mit in der Tabelle der Oberklasse oder aber dass die Unterklassen jeweils in einer eigenen Tabelle abgebildet werden. Die Unterklassen erben die ObjektID der Oberklassen und die Beziehung zwischen den Tabellen wird über Fremdschlüssel hergestellt. Die an der Vererbung beteiligten Klassen werden jeweils in eine eigene Tabelle abgebildet.

4.3.3.8Assoziationen

Durch Castor JDO werden alle gängigen Assoziationen unterstützt. Die verwendeten Assoziationen wurden schon im Abschnitt 3.2.3.3 vorgestellt, so dass hier nur die Probleme, die während der Verwendung auftraten, erläutert werden. Werden in einem Objekt in mehreren Datentypen jeweils die gleichen Objekte verwendet, so besteht bei Castor JDO die Möglichkeit, die beiden Fremdschlüssel auf nur eine Spalte abzubilden.

Da Castor keine Abbildungsregel zur Handhabung der `java.util.Properties` zur Verfügung stellt, wurde eine Hilfsklasse `Property` erstellt, die die Schlüssel und zugehörigen Werte verwaltet. Eine weitere Hilfsklasse `PropertyHelper`, die statische Methoden zur Verfügung stellt, übernimmt das Mapping zwischen den in der Klasse `Primary` befindlichen `java.util.Properties` und der Hilfsklasse `Property`. Die Funktionsweise der beiden Hilfsklassen wird in der beiliegenden Java-Doc erläutert.

4.3.3.9Komposition

Durch Implementierung der Schnittstelle `Persistent` wird ein Callback-Mechanismus zur Verfügung gestellt, durch den das kaskadierende Löschen manuell implementiert werden kann. Diese Vorgehensweise wird im Abschnitt 3.2.3.1 erläutert.

4.3.3.10Abfragesprache

Castor definiert eine eigene Abfragesprache OQL, um auf Objekte in der Datenbank zu zugreifen. Es ist eine Mischung zwischen SQL- und einer objektorientierten Syntax. Es folgt ein Beispiel, bei dem alle `Primary`-Objekte aus der Datenbank gelesen werden, deren Attributwert `boolValueP=true` ist.

```
Query query;
Query = db.getOQLQuery(  SELECT p FROM castor.Primary p
                              WHERE p.boolValueP = true  );
QueryResult result = query.execute();
while(result.hasMore())
{
   Primary prim = (Primary) result.next();
}
```

Der Methode `getOQLQuery(...)` wird einen `String` übergeben, der die Anfrage an die Datenbank darstellt. Es wird direkt mit den Objekten gearbeitet, und durch die Punktnotation auf die zugehörigen Attribute der Objekte zugegriffen. Als Ergebnis wird

eine Enumeration zurückgeliefert, die mit einer einfachen Schleife durchlaufen werden kann.

4.3.3.11 Unterstützte Systemumgebung

Castor ist in der eigenständigen und eingebetteten Umgebung lauffähig (siehe Abschnitt 2.2.1.1).

4.4 Solarmetric Kodo JDO V2.3.4

Dieser Abschnitt beschreibt die Ergebnisse und die aufgetretenen Probleme während der Evaluierung der JDO-Implementierung Kodo JDO V2.3.4.

4.4.1 Automatisches Erstellen der zugehörigen Tabellen

Getestet wurde das Hilfswerkzeug `SchemaTool`. Gestartet wird es wie folgt:

```
java com.solarmetric.kodo.impl.jdbc.schema.SchemaTool –properties kododiplom/kodo.properties –outfile
kodoDiplom.sql –action refresh kododiplom/kododiplom.jdo
```

Durch Angabe des Parameters `properties` wird die zuvor erstelle Konfigurationsdatei „`kodo.properties`[1]“ ausgelesen. Aus dieser werden alle datenbankspezifischen Informationen ausgelesen, u.a. die Angabe des verwendeten Tabellentyp `innoDB` bei MySQL. Der Parameter `outfile` gibt die Datei an, in die das SQL-Skript geschrieben werden soll. Wird der Parameter `action` auf `refresh` gesetzt so werden alle Änderungen, die im übergebenen Persistenz-Deskriptor durchgeführt wurden, erkannt und neue Tabellen bzw. Spalten erstellt bzw. entfernt. Werden, wie im erstellten Persistenz-Deskriptor „`kododiplom.jdo`“, bestimmte Abbildungen weggelassen[2], so verwendet Kodo bei der Erstellung der Spaltennamen eigene Bezeichner. Wird dies nicht gewünscht, so muss ein explizites Mapping dieser Datentypen im Persistenz-Deskriptor vorgenommen werden, so dass eigene Spaltenbezeichner definiert werden können. Bei Verwendung von Datentypen, die die Schnittstelle `Map` implementieren, werden jeweils eigene Tabellen erstellt, die die Schlüssel, zugehörigen Werte und den Fremdschlüssel besitzen, erstellt. Das automatisch erstellte SQL-Skript wies keine Fehler auf und wurde verwendet.

4.4.2 Automatische Generierung des Persistenz-Deskriptors

Durch das Hilfswerkzeug `JDOMetaDataTool` werden durch die Reflection-API die zu den Java-Klassen gehörenden Persistenz-Deskriptoren erstellt. Dies geschieht beispielsweise für die Klasse `AnotherClass` wie folgt:

```
java com.solarmetric.rd.kodo.meta.JDOMetaDataTool –properties kododiplom/kodo.properties –file anotherClass.jdo
kododiplom/AnotherClass.java
```

Dadurch wird das Grundgerüst des Persistenz-Deskriptors automatisch erzeugt. Dies wurde nicht weiter untersucht, da der Persistenz-Deskriptor manuell erstellt wurde.

---

[1] Siehe Abschnitt 3.2.4

[2] Siehe [Russ02, S. 125]: dort werden die Regeln, die ein automatisches Mapping bestimmter Datentypen, die automatisch durch die JDO-Implementierung durchgeführt werden, definiert.

4.4.3Weitere Hilfsprogramme

Wird das eigene Identitätenkonzept verwendet, so können durch das Hilfsprogramm `ApplicationIdTool` die zu den speichernden Klassen gehörenden PrimaryKey-Klassen[3] automatisch erstellt werden. Dies wurde nicht durchgeführt, da die PrimaryKey-Klassen des intelliBO-Prototypes übernommen wurden.

4.4.4Integrationsaufwand für bereits existierende Klassen

Kodo nutzt einen Bytecode-Enhancer. Dieser wird durch Aufruf von:

java com.solarmetric.kodo.enhance.JDOEnhancer -properties kododiplom/kodo.properties kododiplom/*.class

gestartet. In Abschnitt 4.1.2 erfolgte eine Erläuterung dieses Vorganges.

4.4.5Performance und Ressourcenverbrauch

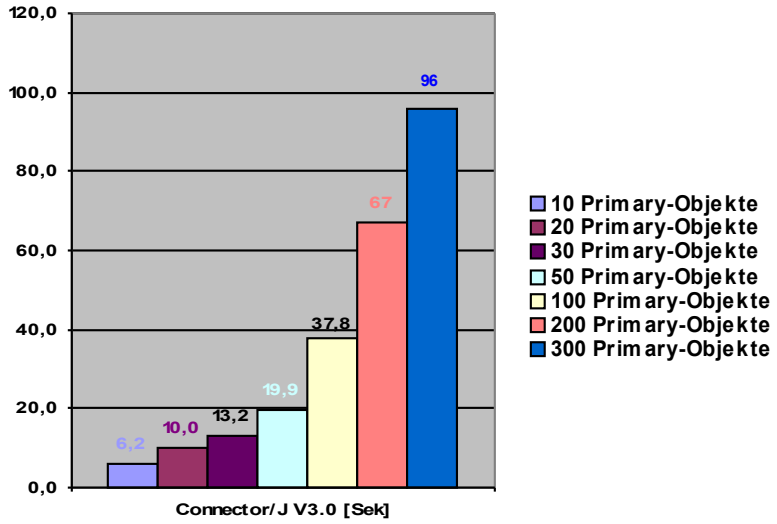
Unter Verwendung der Datenbank MySQL wurden die Tests mit den beiden JDBC-Treibern ([MaMa02] und [Conn03]) durchgeführt. Da bei Benutzung des MM-JDBC-Treibers ein fehlerhaftes Mapping des primitiven Datentyps `boolean` festgestellt wurde, d.h. `false` und `true` wurden jeweils auf 0 in der Datenbank abgebildet, wurde letztendlich nur der JDBC-Treiber Connector/J V3.0 verwendet.

4.4.5.1Insert-Operation

n `Primary`-Objekte werden aus einer `ArrayList` in die Datenbank MySQL eintragen:

Connector/J V3.0:

Anzahl	[ms]
10	6229
20	9995
30	13209
50	19850
100	37787
200	66990
300	95733



Rollback:

[A]: Zeit in [ms], in denen die Hauptobjekte instanziiert, in einer `ArrayList` zwischengespeichert und der Methode `makePersistentAll(List)` übergeben wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]

[3] In Abschnitt 2.2.1.4 wurden die Regeln für die Erstellung der PrimaryKey-Klassen eingeführt

10	2063	10
20	2214	10
30	2333	20
50	2464	100
100	3015	130
200	3806	150
300	4156	180

Fazit:

Durch Aufruf der Methode `makePersistentAll(list)` wechseln alle sich in der `List` befindlichen Objekte vom Zustand transient in den Zustand persistent-new. Durch Aufruf der Methode `commit()` wird der komplette Objektgraph durchlaufen, und alle von diesen Objekten aus referenzierten Objekte (Eigenschaft: Persistenz durch Erreichbarkeit[1]) als persistent deklariert und in der zugrunde liegenden Datenbank gespeichert. Nun wechseln die abgespeicherten Objekte in den Zustand hollow. Wird die aktuelle Transaktion durch Aufruf der Methode `rollback()` abgebrochen, so wechselt dass sich im Zustand persistent-new befindliche Objekt wieder in den Zustand transient. In Relation zu der Anzahl der zu speichernden Objekte ist eine gute Laufzeit zu erkennen.

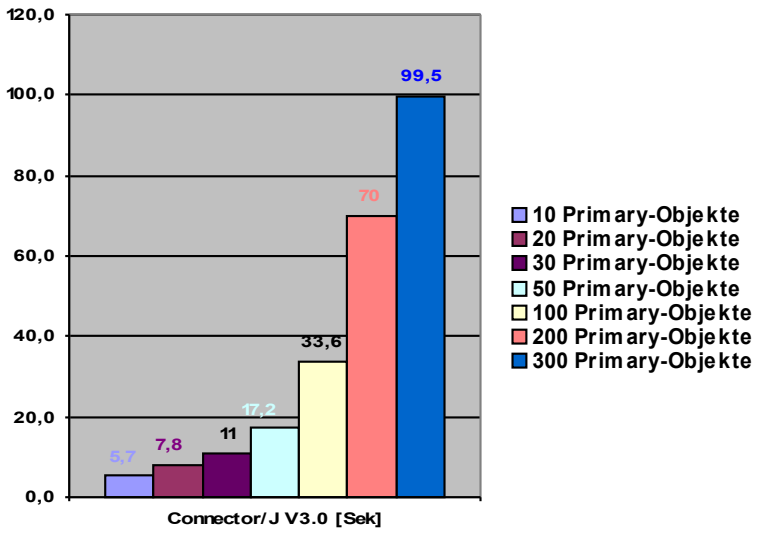
4.4.5.2Delete-Operation

Die n sich im Speicher befindlichen `Primary`-Objekte wurden innerhalb einer Transaktion gelöscht.

n `Primary`-Objekte werden in der Datenbank gelöscht:

Connector/J V3.0:

Anzahl	[ms]
10	5669
20	7822
30	10917
50	17246
100	33550
200	69764
300	99459



Rollback:

[A]: Zeit in [ms], in der die sich im Speicher befindlichen Primary-Objekte der Methode `deletePersistentAll(List)` übergeben wurden

[1] Der Begriff „Persistenz durch Erreichbarkeit“ wurde in Abschnitt 2.2.1.3 eingeführt

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	10	10
20	10	10
30	10	20
50	10	20
100	10	20
200	10	20
300	10	30

Fazit:

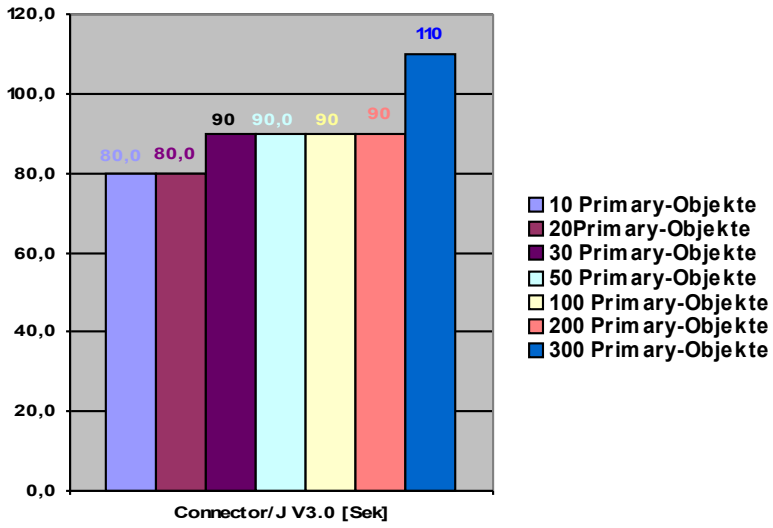
Durch Aufruf der Methode `deletePersistentAll(List)` wird den Objekten der Zustand persistent-clean entzogen und sie wechseln in den Zustand persistent-deleted. Wird nun die aktuelle Transaktion erfolgreich durch Aufruf der Methode `commit()` beendet, so wechseln die sich in der `List` befindlichen Objekte in den Zustand transient. Wird hingegen die aktuelle Transaktion durch Aufruf der Methode `rollback()` abgebrochen, so wechseln die Objekte von dem Zustand persistent-deleted erneut in den Zustand hollow. Auch hier ist in Relation zu der Anzahl der zu speichernden Objekte eine gute Laufzeit zu erkennen.

4.4.5.3Query-Operation

n `Primary`-Objekte werden aus der Datenbank gelesen und in einer `ArrayList` zwischengespeichert:

Connector/J V3.0:

Anzahl	[ms]
10	80
20	80
30	90
50	90
100	90
200	90
300	110



Rollback:

[A]: Zeit in [ms], in der eine Abfrage an die Datenbank erfolgte und alle Hauptobjekte in einer `ArrayList` gespeichert wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]

10		
20		
50		
100		
200		
300		

Fazit:

Wird ein JDBC-Treiber verwendet, der die JDBC-Spezifikation V2.0 oder höher implementiert, so ist es nicht möglich in der Konfigurationsdatei durch setzen des Wertes `com.solarmetric.kodo.DefaultFetchThreshold` auf `-1` das „lazy loading“ auszuschalten (siehe [Kodo03, S. 83]). D.h. das Hauptobjekt liegt initialisiert im Speicher vor. Wird auf die von diesem aus referenzierten Objekte zugegriffen, so werden diese Werte bei Bedarf aus der Datenbank gelesen. Nach Ausführung der Query-Operation wechseln die ausgelesenen Objekte von dem Zustand hollow in den Zustand persistent-clean. Findet nun keine Änderung an den ausgelesenen Objekten statt, so wechselt der Zustand nach Aufruf der Methode `commit()` oder `rollback()` erneut in den Zustand hollow.

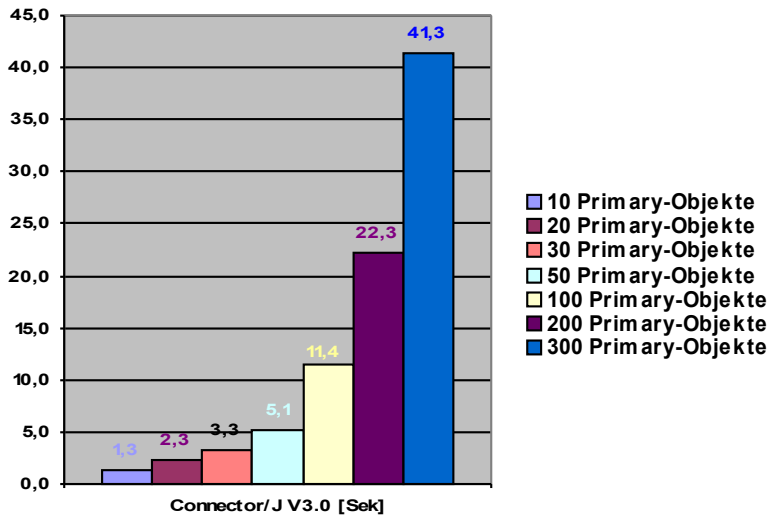
4.4.5.4Update-Operation

Allen Objekten der Klasse `SecondaryFirst` wurde ein neuer Bezeichner zugewiesen, d.h. die sich im Speicher befindlichen Objekte wurden verändert und somit mit der Datenbank synchronisiert:

Die Anzahl bezieht sich auf die `Primary`-Objekte. Die daraus resultierende Anzahl von `SecondaryFirst`-Objekten steht jeweils in Klammern.

Connector/J V3.0:

Anzahl	[ms]
10 (200)	1292
20 (400)	2274
30 (600)	3295
50 (1000)	5087
100 (2000)	11417
200 (4000)	22364
300 (6000)	41342



Rollback:

[A]: Zeit in [ms], in der die Änderung durchgeführt wurde

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	1292	30
20	1843	35

50	4417	45
100	8713	55
200	20380	70

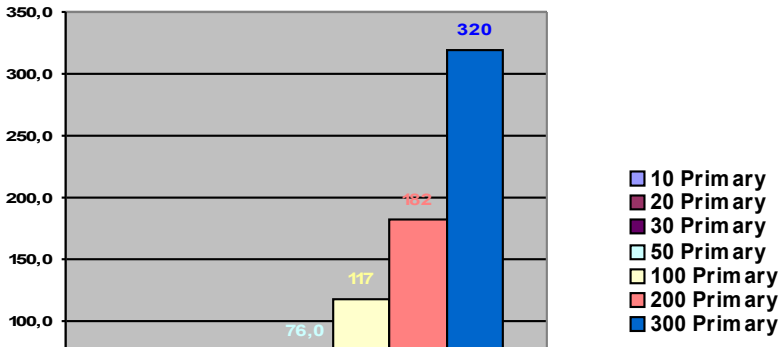
Fazit:

Durch Aufruf der Methode `setString(stringValue)` wird allen `SecondaryFirst`-Objekten ein neuer Bezeichner zugewiesen. Dadurch wechseln die Objekte in den Zustand persistent-dirty. Nun liegt ein inkonsistenter Zustand zwischen den Werten in den Objekten und der Datenbank vor. Wird nun die aktuelle Transaktion durch Aufruf der Methode `commit()` erfolgreich beendet, so wechselt der Zustand der Objekte von persistent-dirty in den Zustand hollow, und die Änderungen werden in der Datenbank abgespeichert. Wird die aktuelle Transaktion durch Aufruf von `rollback()` abgebrochen, so findet der gleiche Zustandswechsel statt, aber die Änderungen werden verworfen, d.h. sie werden nicht in der Datenbank abgespeichert. Auch hier ist in Relation zu der Anzahl der zu speichernden Objekte eine lineare Laufzeit zu erkennen.

4.4.5.5Speicherverbrauch

Es folgt eine Auflistung des Speicherverbrauchs, den der Testfall (obige Operationen) bei Verwendung der Datenbank MySQL benötigt. Die Angaben wurden im Taskmanager abgelesen:

Anzahl	[MB]
10	26
20	35
30	51
50	76
100	117
200	182
300	320



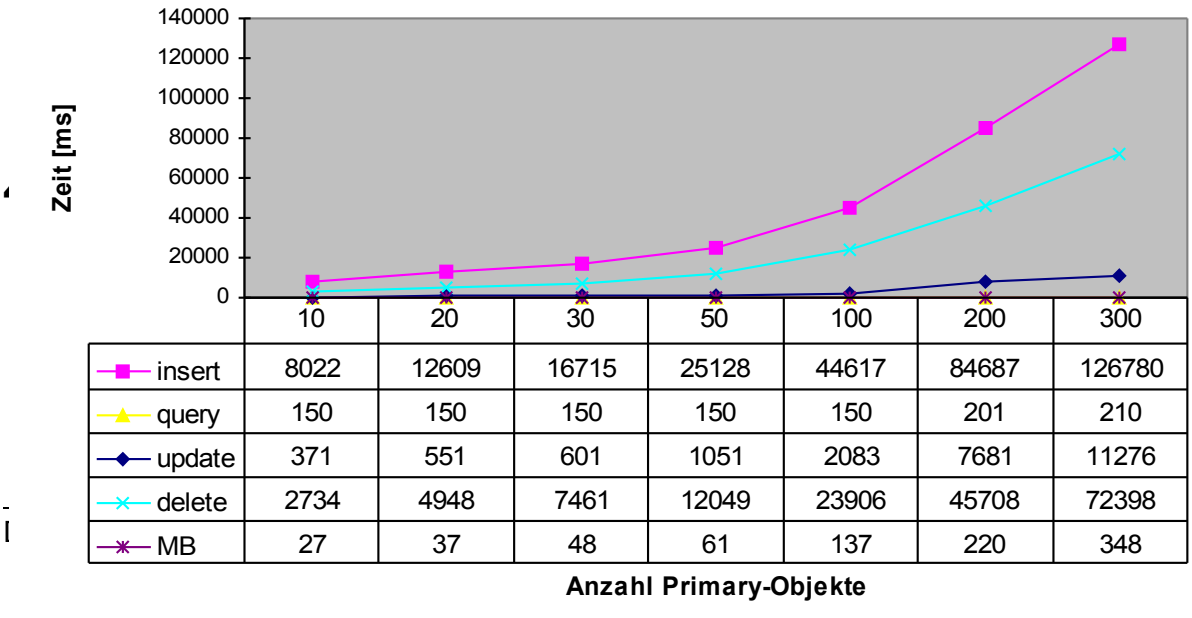
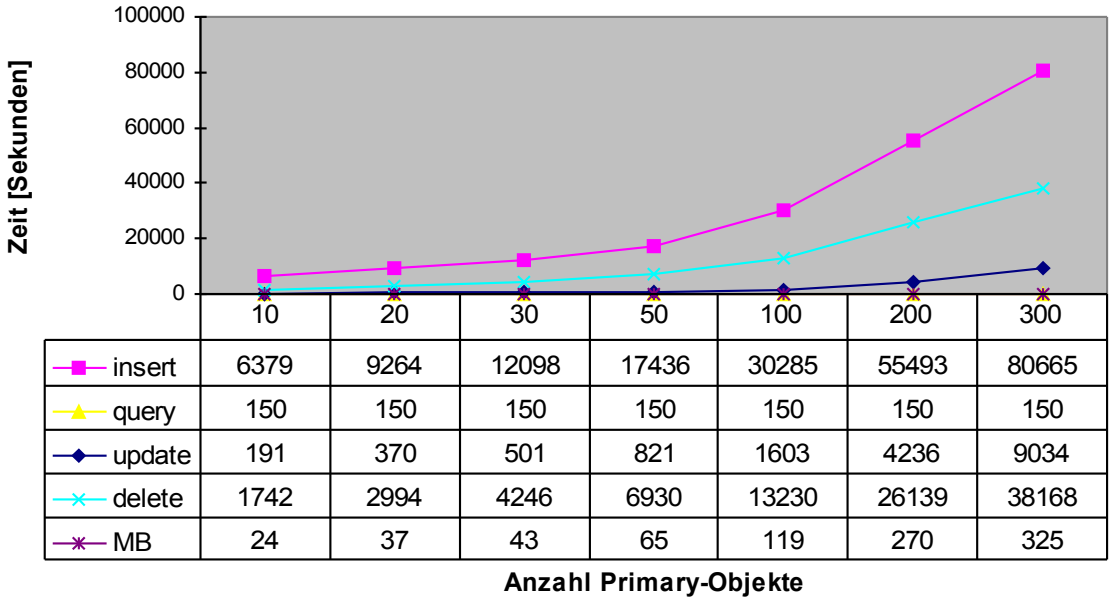


Abbildung 17„Performance-Ergebnisse unter Verwendung der Datenbank HSQL im Server-Modus“



	10	20	30	50	100	200	300
insert	6379	9264	12098	17436	30285	55493	80665
query	150	150	150	150	150	150	150
update	191	370	501	821	1603	4236	9034
delete	1742	2994	4246	6930	13230	26139	38168
MB	24	37	43	65	119	270	325

Abbildung 18„Performance-Ergebnisse unter Verwendung der Datenbank HSQL im embedded-Modus“

4.4.6Vererbung

Die Grundeinstellung von KODO ist, dass alle an einer Vererbung beteiligten Klassen in einer Tabelle abgebildet werden. Dies kann durch Setzen über die Option `com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping` in der Konfigurationsdatei „`kodo.properties`“ auf `false` unterbunden werden. Im Rahmen der Evaluierung wurden alle an der Vererbung beteiligeten Klassen in einer eigenen Tabelle abgebildet. Dazu muss in der zur Oberklasse gehörenden Tabelle eine zusätzliche Spalte aufgenommen werden, in der der vollständige Name der jeweiligen Unterklassen verwaltet wird. Die daraus resultierenden Tabellen müssen den gleichen Spaltennamen für die ObjektID besitzen, wie die Oberklasse. Die Unterklassen erben die ObjektID der Oberklasse und die Beziehung zwischen den Tabellen wird über Fremdschlüssel hergestellt.

4.4.7Assoziationen

Kodo unterstützt alle gängigen Assoziationen. In Abschnitt 3.2.4.2 wurden schon die verwendeten Assoziationen vorgestellt, so dass hier nur die Probleme, die während der Verwendung auftraten, erläutert werden. Werden in einem Objekt mehrere Datentypen, z.B. `List` und `Map` verwendet, die die gleichen Elementtypen verwalten, so muss jeweils eine eigene Spalte für den Fremdschlüssel definiert werden. Werden die Fremdschlüssel jeweils auf nur eine Spalte abgebildet, so funktioniert eine Query auf die `Map` korrekt, aber nicht auf die `List`. Das Problem bei „lazy loading“ wurde schon in Abschnitt 4.4.5.3 erläutert.

4.4.8Komposition

Kodo unterstützt das kaskadierende Löschen bei den 1…1-, 1…N- und N…M-Beziehungen. Dieses kann im Persistenz-Deskriptor aktiviert werden (siehe Abschnitt 3.2.4.2).

4.4.9Graphische Entwicklungsumgebung

Kodo liegt keine eigene graphische Entwicklungsumgebung bei, sondern nur Kommandozeilen-basierte Hilfswerkzeuge, auf die in den vorherigen Abschnitten schon eingegangen wurde. Es besteht aber die Möglichkeit über Plug-Ins Kodo in Produkte anderer Hersteller zu integrieren:

- Apache Ant
- Sourceforge XDoclet
- Borland JBuilder V7 oder höher
- SUN ONE Studio/NetBeans DIE
- Eclipse/WebSphere Studio Integration

4.4.10Abfragesprache(n)

Kodo nutzt als Abfragesprache JDOQL, die schon im Abschnitt 2.2.6 eingeführt wurde.

4.4.11Unterstützte Systemumgebung

Kodo wird in zwei Versionen angeboten. Die Standard-Edition ist in der eigenständigen Umgebung, und die Enterprise-Edition in der eingebetteten Umgebung lauffähig (siehe Abschnitt 2.2.1.1).

4.5 Libelis Lido V1.4

4.5.1 Automatisches Erstellen der zugehörigen Tabellen

Lido liegt ein Kommandozeilen-basiertes Hilfswerkzeug zur automatischen Erstellung der benötigten Tabellen bei. Gestartet wird es wie folgt:

java com.libelis.lido.DefineSchema -properties c:\lidodiplom\lido –metadata c:\lidodiplom\lidodiplom.jdo –sql c:\lidodiplom\lidoSQL.sql

Die zuvor erstellte Konfigurationsdatei „`lido.properties`“ wird durch Angabe des Parameters `properties` ausgelesen. Aus dieser werden alle datenbankspezifischen Informationen ausgelesen. Duch den Parameter `metadata` wird der Persistenz-Deskriptor und durch `sql` die Datei, in die das SQL-Skript gespeichert werden soll, angegeben. Werden, wie im erstellten Persistenz-Deskriptor „`lidodiplom.jdo`“, bestimmte Abbildungen weggelassen[1], so verwendet Lido bei der Erstellung der Spaltennamen eigene Bezeichner. Wird dies nicht gewünscht, so muss ein explizites Mapping dieser Datentypen im Persistenz-Deskriptor vorgenommen werden, so dass eigene Spaltenbezeichner definiert werden können. Werden Datentypen verwendet, die die Schnittstellen `Map` oder `List` implementieren, so werden jeweils eigene Tabellen erstellt. Die Tabelle, die die `List` repräsentiert, besitzt Spalten, die die Fremdschlüssel und die Position der Elemente in der `List`, speichern. In der zur `Map` gehörenden Tabelle werden die Fremdschlüssel und der Schlüssel der `Map` verwaltet.

Lediglich bei Verwendung des Tabellentyps `innoDB` bei MySQL, muss dies manuell in das erstellte SQL-Skript lidoSQL.sql eingefügt werden.

4.5.2 Integrationsaufwand für bereits existierende Klassen

Lido nutzt einen Bytecode-Enhancer. Dieser wird durch Aufruf von:

java com.libelis.lido.Enhance -metadata c:\lidodiplom\lidodiplom.jdo

gestartet. In Abschnitt 4.1.2 erfolgte eine Erläuterung dieses Vorganges.

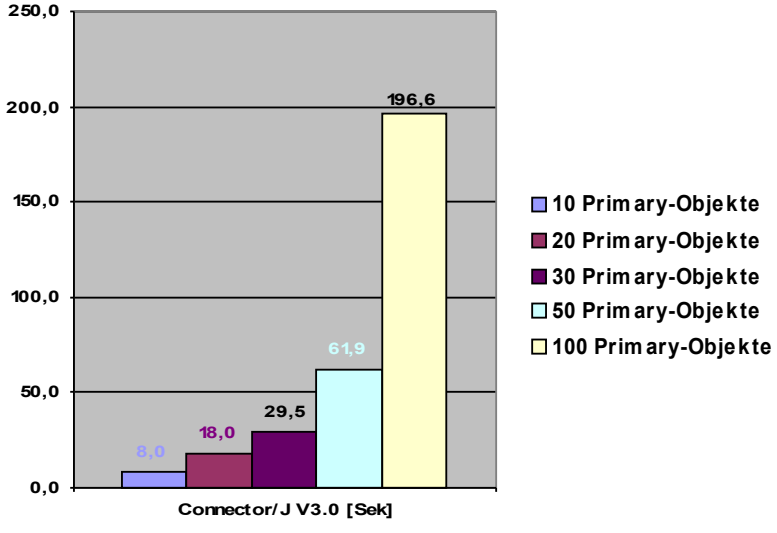
4.5.3 Performance und Ressourcenverbrauch

Unter Verwendung der Datenbank MySQL wurden die Tests mit dem JDBC-Treiber [Conn03] durchgeführt. Es folgt eine Auflistung der Ergebnisse der einzelnen Operationen.

4.5.3.1 Insert-Operat

n Primary-Objekte werde

Connector/J V3.0:



[1] Siehe [Russ02, S. 125]: dort werd ch die JDO-Implementierung durchgefü

Anzahl	[ms]
10	7962
20	18047
30	29454
50	61893
100	196615

Rollback:

[A]: Zeit in [ms], in denen die Hauptobjekte instanziiert, in einer `ArrayList` zwischengespeichert und der Methode `makePersistentAll(List)` übergeben wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	1382	141
20	2224	220
30	3135	300
50	4748	480
100	7712	801

Fazit:

Durch den Aufruf der Methode `makePersistentAll(list)` wechseln alle sich in der List befindlichen Primary-Objekte vom Zustand transient in den Zustand persistent-new. Durch Aufruf der Methode `commit()` wird der komplette Objektgraph durchlaufen, und alle von diesen Objekten aus referenzierten Objekte (Eigenschaft: Persistenz durch Erreichbarkeit[1]) als persistent deklariert und in der zugrunde liegenden Datenbank gespeichert. Nun wechseln die abgespeicherten Objekte in den Zustand hollow. Wird die aktuelle Transaktion durch Aufruf der Methode `rollback()` abgebrochen, so wechselt dass sich im Zustand persistent-new befindliche Objekt wieder in den Zustand transient. Es ist eine exponentielle Laufzeit zu erkennen.

4.5.3.2Delete-Operation

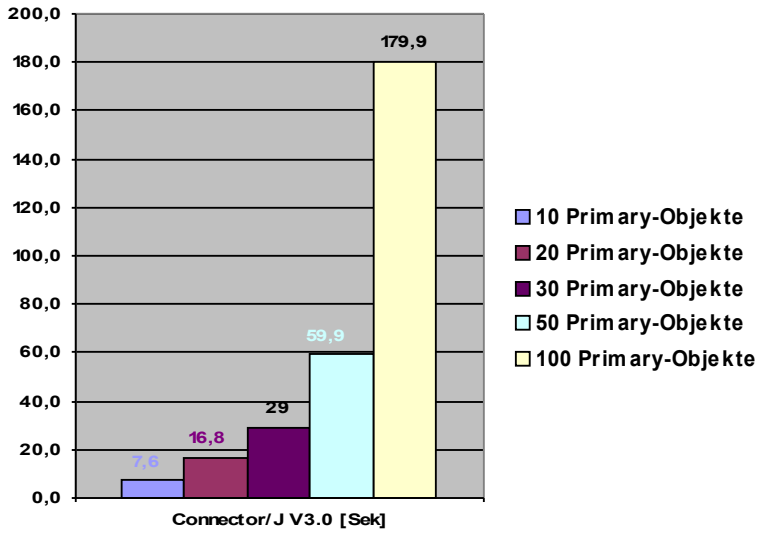
Die n sich im Speicher befindlichen `Primary`-Objekte wurden innerhalb einer Transaktion gelöscht.

n `Primary`-Objekte werden in der Datenbank gelöscht:

Connector/J V3.0:

[1] Der Begriff „Persistenz durch Erreichbarkeit“ wurde in Abschnitt 2.2.1.3 eingeführt

Anzahl	[ms]
10	7552
20	16765
30	28983
50	59850
100	179910



Rollback:

[A]: Zeit in [ms], in der die sich im Speicher befindlichen Primary-Objekte der Methode `deletePersistentAll(obj)` übergeben wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	4416	81
20	7000	141
30	11027	230
50	18528	355
100	35032	661

Fazit:

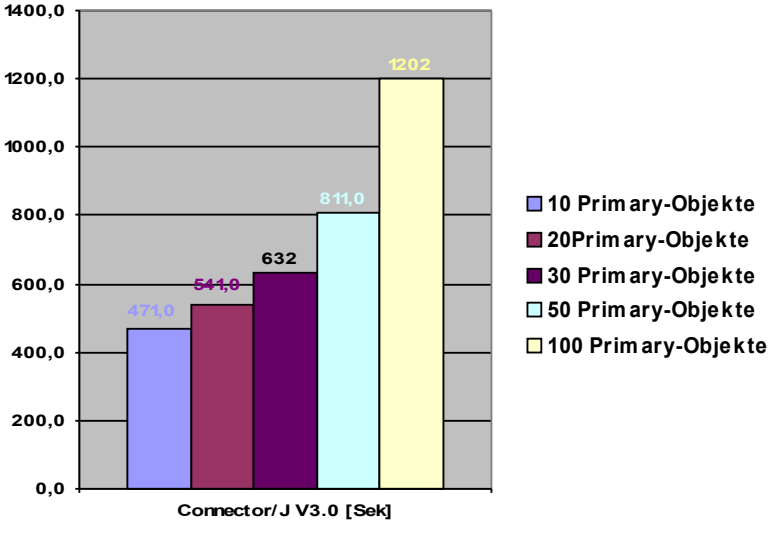
Durch Aufruf der Methode `deletePersistentAll(List)` wird den Objekten der Zustand persistent-clean entzogen und sie wechseln in den Zustand persistent-deleted. Wird nun die aktuelle Transaktion erfolgreich durch Aufruf der Methode `commit()` beendet, so wechseln die sich in der List befindlichen Objekte in den Zustand transient. Wird hingegen die aktuelle Transaktion durch Aufruf der Methode `rollback()` abgebrochen, so wechseln die Objekte von dem Zustand persistent-deleted erneut in den Zustand hollow. Auch
Objekte eine exponentielle

4.5.3.3Query-Operatio

n `Primary`-Objekte werde
zwischengespeichert:

Connector/J V3.0:

Anzahl	[ms]
10	471
20	541
30	632



50	811
100	1202

Rollback:

[A]: Zeit in [ms], in der eine Abfrage an die Datenbank erfolgte und alle Hauptobjekte in einer `ArrayList` gespeichert wurden

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	461	60
20	475	100
30	505	151
50	651	271
100	886	530

Fazit:

Auch Lido unterstützt das Konzept der „trägen Initialisierung“. D.h. nur das Hauptobjekt liegt initialisiert im Speicher vor. Wird auf die von diesem aus referenzierten Objekte zugegriffen, so werden diese Werte erst dann aus der Datenbank gelesen. Nach Ausführung der Query-Operation wechseln die ausgelesenen Objekte von dem Zustand hollow in den Zustand persistent-clean. Findet nun keine Änderung an den ausgelesenen Objekten statt, so wechselt der Zustand nach Aufruf der Methode `commit()` oder `rollback()` erneut in den Zustand hollow.

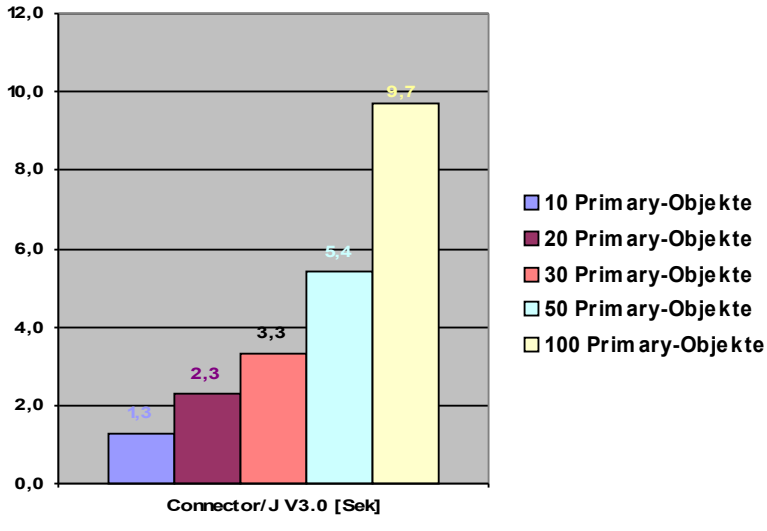
4.5.3.4Update-Operation

Allen Objekten der Klasse `SecondaryFirst` wurde ein neuer Bezeichner zugewiesen, d.h. die sich im Speicher befindlichen Objekte wurden verändert und somit mit der Datenbank synchronisiert:

Die Anzahl bezieht sich auf `SecondaryFirst`-Objekte

Connector/J V3.0:

Anzahl	[ms]
10 (200)	1252
20 (400)	2264
30 (600)	3295
50 (1000)	5358



100 (2000)	9764

Rollback:

[A]: Zeit in [ms], in der die Änderung durchgeführt wurde

[B]: Zeit in [ms], die der rollback-Vorgang in Anspruch genommen hat

Anzahl	[A]	[B]
10	1022	50
20	1853	110
50	4227	280
100	7621	561

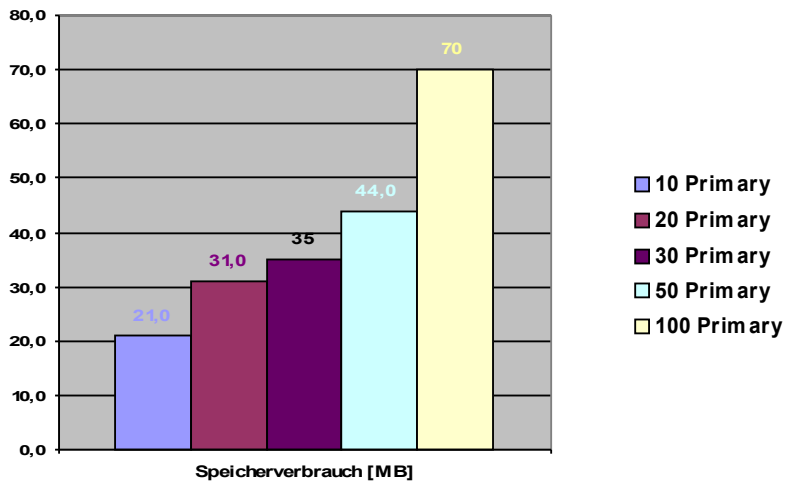
Fazit:

Durch Aufruf der Methode `setString(stringValue)` wird allen `SecondaryFirst`-Objekten ein neuer Bezeichner zugewiesen. Dadurch wechseln die Objekte in den Zustand persistent-dirty. Nun liegt ein inkonsistenter Zustand zwischen den Werten in den Objekten und der Datenbank vor. Wird nun die aktuelle Transaktion durch Aufruf der Methode `commit()` erfolgreich beendet, so wechselt der Zustand der Objekte von persistent-dirty in den Zustand hollow, und die Änderungen werden in der Datenbank abgespeichert. Wird die aktuelle Transaktion durch Aufruf von `rollback()` abgebrochen, so findet der gleiche Zustandswechsel statt, aber die Änderungen werden verworfen, d.h. sie werden nicht in der Datenbank abgespeichert. Hier ist in Relation zu der Anzahl der zu speichernden Objekten eine lineare Laufzeit zu erkennen.

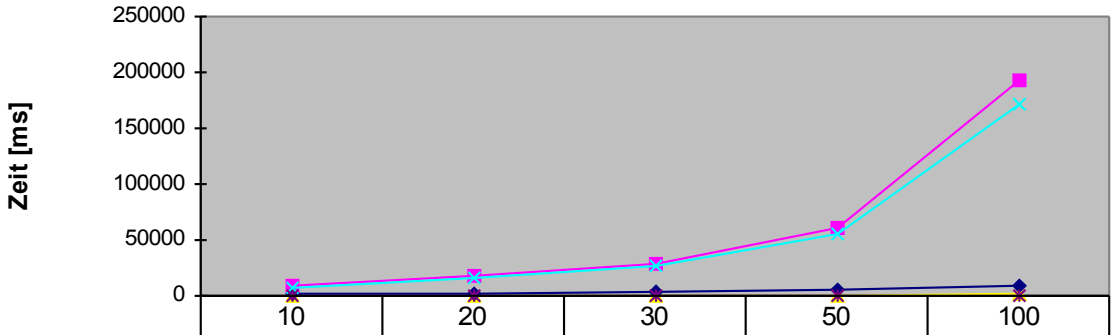
4.5.3.5Speicherverbrauch

Es folgt eine Auflistung des Speicherverbrauchs, den der Testfall (obige Operationen) bei Verwendung der Datenbank MySQL benötigt. Die Angaben wurden im Taskmanager abgelesen:

Anzahl	[MB]
10	21
20	31
30	35
50	44
100	70



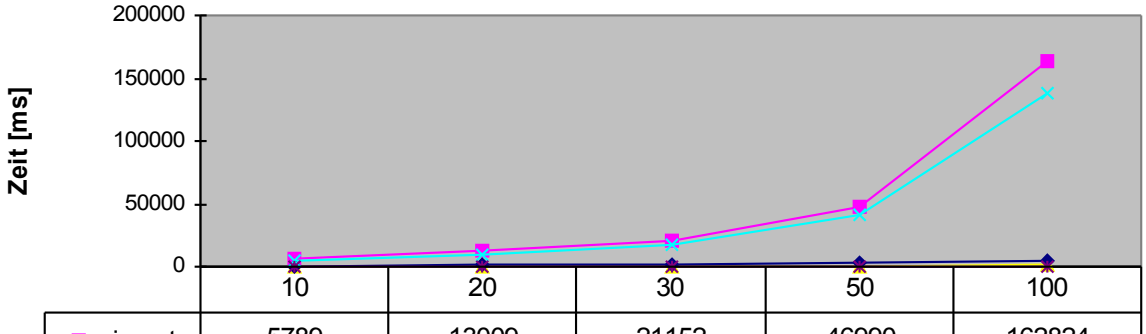
4.5.3.6Performance unter Verwendung der Datenbank HSQL



	10	20	30	50	100
insert	8121	17366	29283	61141	192838
query	480	521	621	781	1182
update	1112	2013	2865	4607	8663
delete	6821	15944	26529	56074	171287
MB	21	25,5	33	45	72

Anzahl Primary-Objekte

Abbildung 19„Performance-Ergebnisse unter Verwendung der Datenbank HSQL im Server-Modus“



	10	20	30	50	100
insert	5789	13009	21152	46990	162824
query	470	541	611	791	1212
update	711	1192	1723	2685	4767
delete	4016	8963	17716	40641	138888
MB	25	37	40,5	52	82

Anzahl Primary-Objekte

Abbildung 20„Performance-Ergebnisse unter Verwendung der Datenbank HSQL im embedded-Modus“

4.5.4Vererbung

Auch Lido unterstützt das Konzept der Vererbung. Im Gegensatz zu den anderen untersuchten JDO-Implementierungen wird nicht jede an der Verebung beteiligte Klasse in einer eigenen Tabelle abgebildet. Sondern alle an der Vererbung beteiligten Klassen bzw. deren Attributwerte werden in der Tabelle der am weitesten abgeleiteten Klasse gespeichert.

4.5.5Assoziationen

Lido unterstützt alle gängigen Assoziationen. In Abschnitt 3.2.5.2 wurden schon die verwendeten Assoziationen vorgestellt.

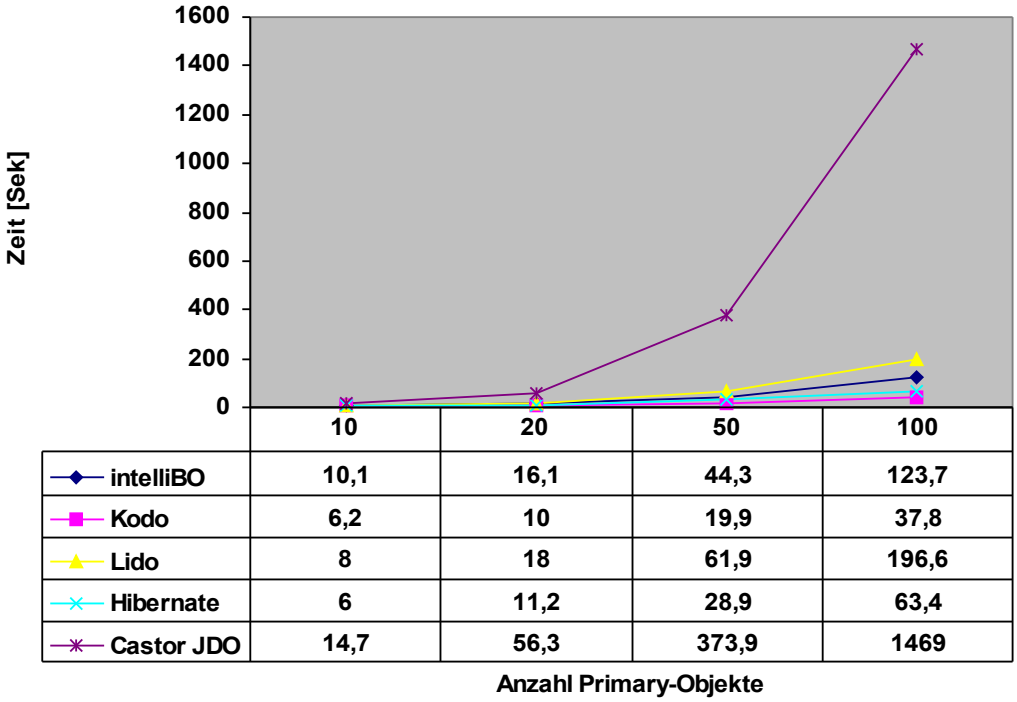
4.5.6Komposition

Das kaskadierende Löschen wird in der aktuellen Version nicht automatisch unterstützt. In der nächsten Version soll dieser Mechanismus aufgenommen werden. Durch Implementierung der Schnittstelle `InstanceCallbacks` (siehe Abschnitt 2.2.4.5) muss dies manuell implementiert werden.

5. Zusammenfassung

In diesem Abschnitt werden die untersuchten JDO-Implementierungen und die objektrelationalen Abbildungswerkzeuge anhand der Datenbank-Operationen insert, delete, query und update gegenübergestellt. Es werden jeweils die gleiche Anzahl an `Primary`-Objekten unter Verwendung der Datenbank MySQL dargestellt.

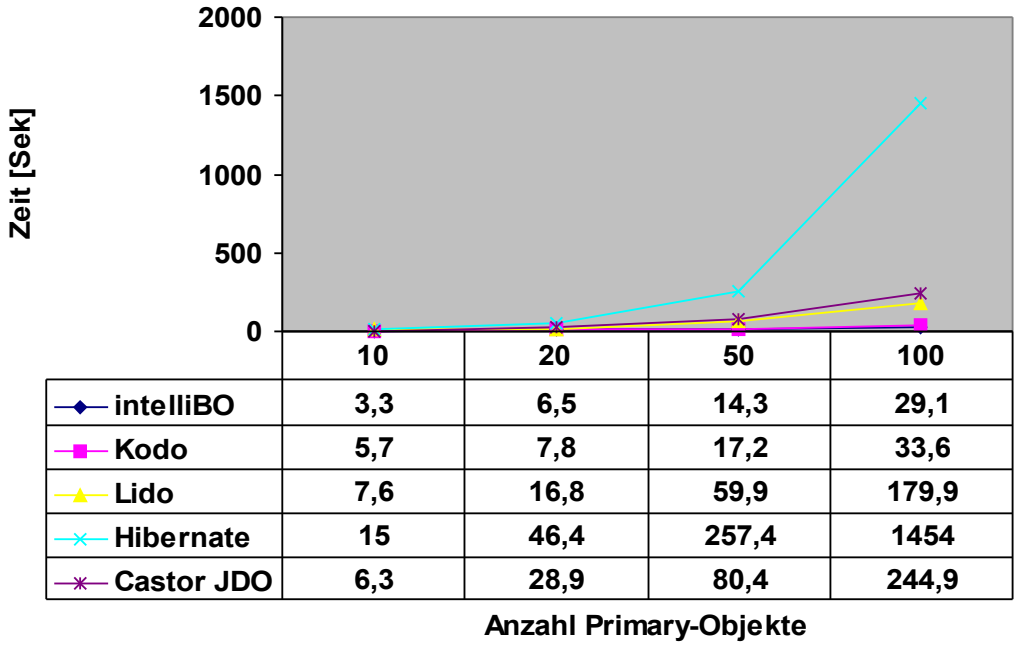
Die insert-Operation:



	10	20	50	100
intelliBO	10,1	16,1	44,3	123,7
Kodo	6,2	10	19,9	37,8
Lido	8	18	61,9	196,6
Hibernate	6	11,2	28,9	63,4
Castor JDO	14,7	56,3	373,9	1469

Anzahl Primary-Objekte

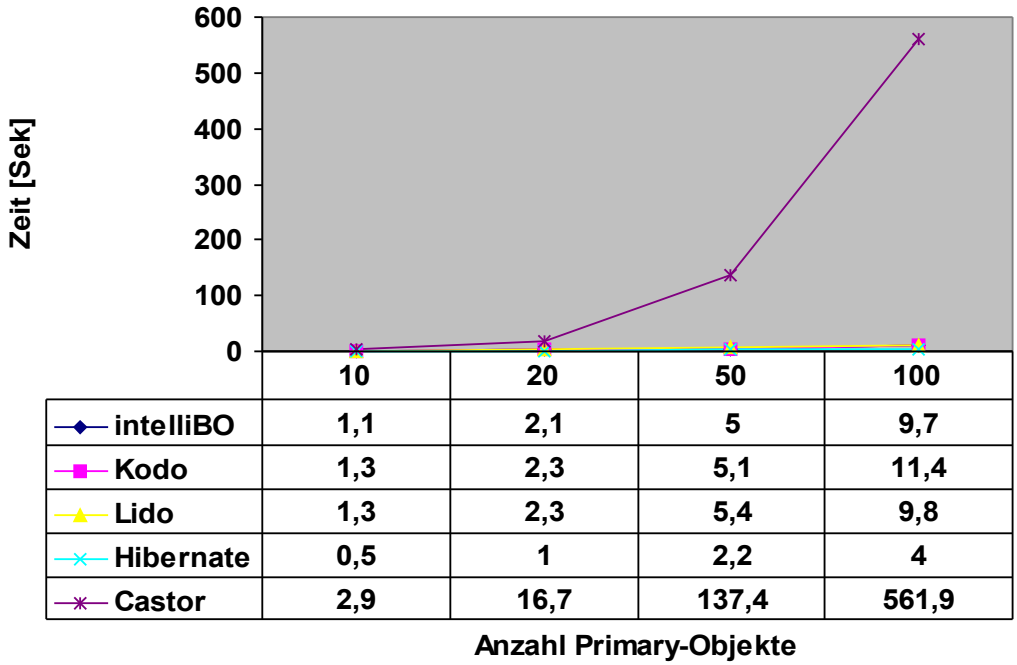
Die delete-Operation:



	10	20	50	100
intelliBO	3,3	6,5	14,3	29,1
Kodo	5,7	7,8	17,2	33,6
Lido	7,6	16,8	59,9	179,9
Hibernate	15	46,4	257,4	1454
Castor JDO	6,3	28,9	80,4	244,9

Anzahl Primary-Objekte

Die update-Operation:

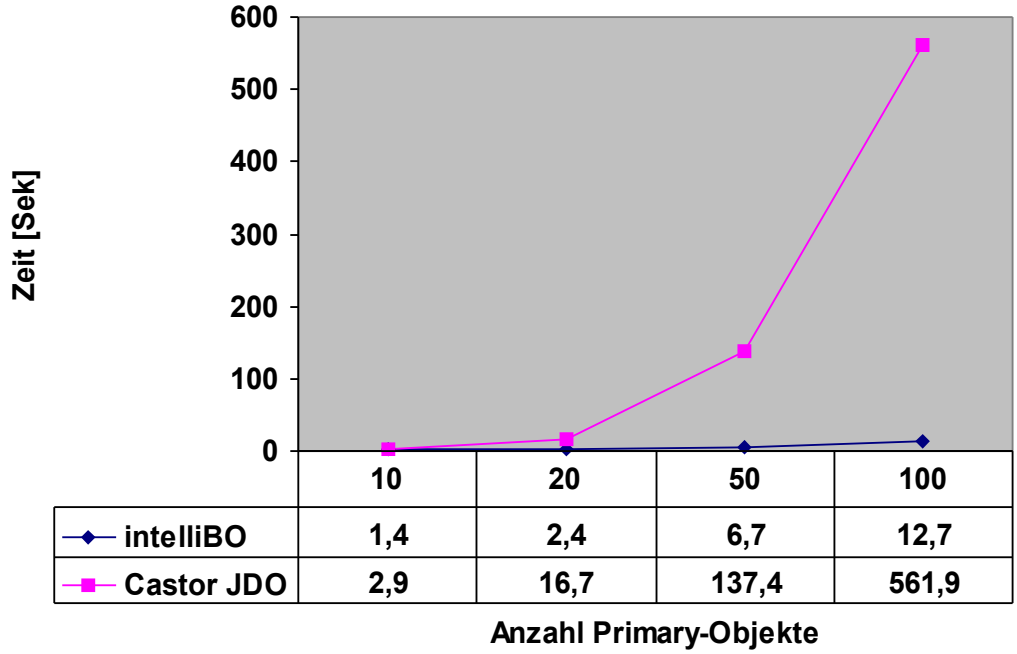


	10	20	50	100
intelliBO	1,1	2,1	5	9,7
Kodo	1,3	2,3	5,1	11,4
Lido	1,3	2,3	5,4	9,8
Hibernate	0,5	1	2,2	4
Castor	2,9	16,7	137,4	561,9

Die query-Operation:

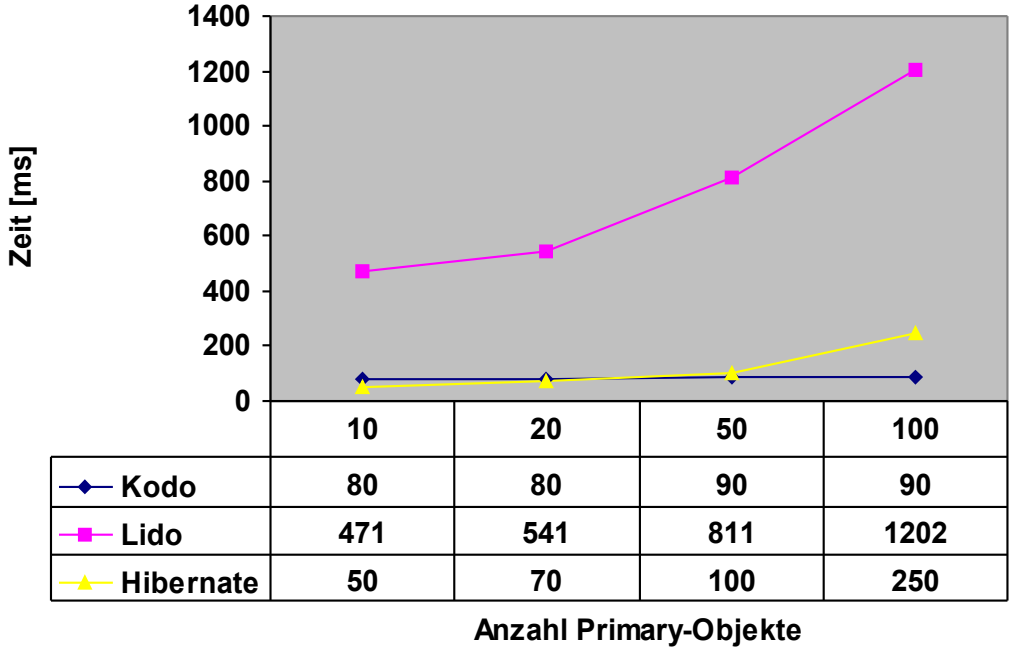
Da Kodo, Lido und Hibernate das Konzept des „lazy loading“ verfolgen, erfolgt eine getrennte Gegenüberstellung der query-Operation:

ohne „lazy loading“:

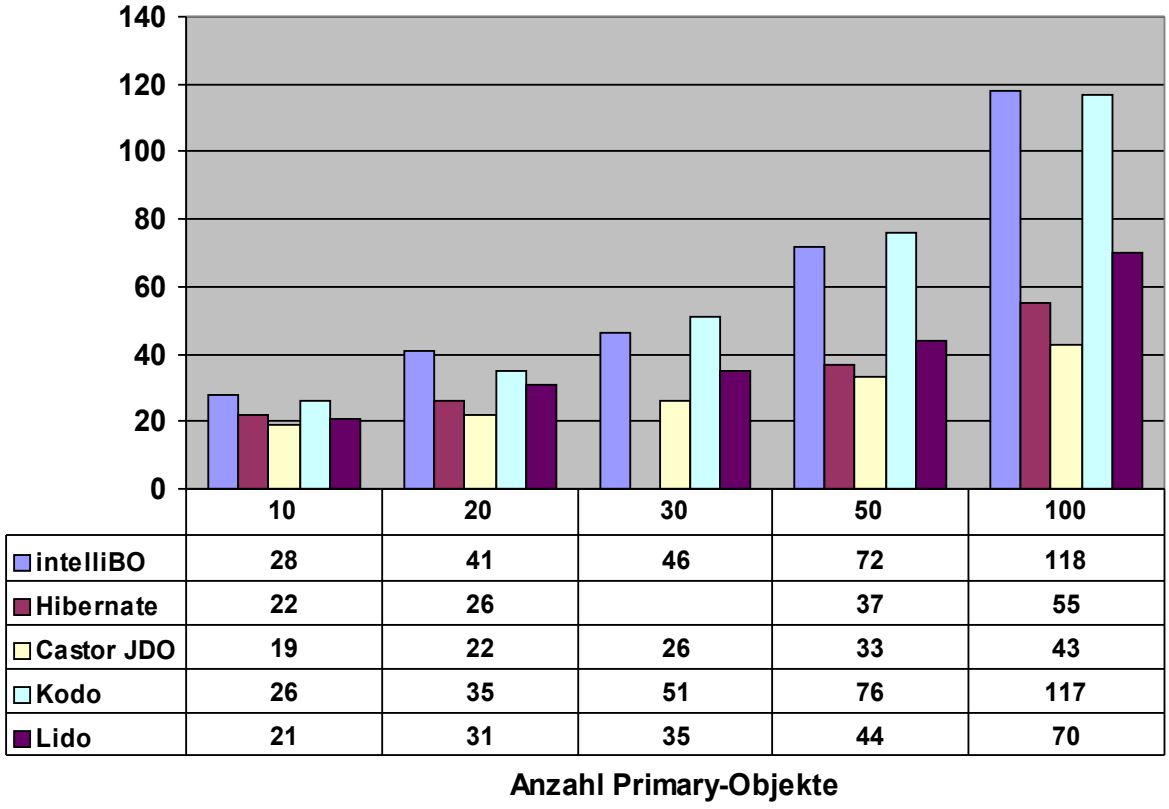


	10	20	50	100
intelliBO	1,4	2,4	6,7	12,7
Castor JDO	2,9	16,7	137,4	561,9

mit „lazy loading“:



Speicherverbrauch [MB]:



Ziel dieser Diplomarbeit war es verschiedene JDO-Implementierungen und objektrelationale Abbildungswerkzeuge anhand von Selektions- und

Evaluierungskriterien in Bezug auf ein Projekt im Leittechnikumfeld der Abteilung Power Generation der Siemens AG Erlangen zu untersuchen.

Zu Beginn wurde ein Prototyp erstellt, der die Komplexität der zu speichernden Objekte repräsentiert. Anhand dessen wurden drei JDO-Implementierungen (intelliBO, Kodo und Lido) und zwei Objekt-Relational-Abbildungswerkzeuge (Hibernate und Castor JDO) in Bezug auf die in Abschnitt 2.5 genannten Kriterien evaluiert.

Anhand der oben aufgeführten Diagrammen, welche die Ergebnisse der Datenbank-Operationen insert, delete, update und query darstellen, ist zu erkennen, dass sich keine untersuchte Technik, bezogen auf den in Abschnitt 3 genannten Anwendungsfall, heraus kristallisiert.

Weist eine Technik bei ein oder mehreren Datenbank-Operationen eine akzeptable Laufzeit auf, so steht dem bei einer anderen Datenbank-Operation eine exponentielle Laufzeit entgegen.

Interessant ist die Feststellung, dass die beiden objektrelationalen Abbildungswerkzeuge die Ausnutzung des Speichers besser im Griff haben als die drei JDO-Implementierungen. So benötigen bei gleicher Anzahl der zu speichernden Objekte die JDO-Implementierungen meistens doppelt so viel Arbeitsspeicher als die objektrelationalen Abbildungswerkzeuge.

Im Verlauf der Diplomarbeit kamen neuere Versionen der untersuchten Produkte heraus, in denen Fehler behoben und in die Erweiterungen einbezogen worden sind. Auf Grund dessen ist zu erkennen, dass die JDO-Implementierungen noch nicht ausgereift sind.

Als positive Anmerkung ist zu sagen, dass sich nach einer angemessenen Einarbeitungszeit die jeweilige Technik schnell einsetzen lässt. Sehr gut erfolgte auch die Umsetzung der „transparenten Persistenz“, der Anwender benötigt keine SQL oder JDBC-Kenntnisse. Werden nicht viele komplexe Objekte innerhalb einer Transaktion gespeichert, so treten keine Speicherprobleme auf.

6. Anhang

6.1 Abbildungsverzeichnis

2.2.1.1: „JDO in einer eigenständigen Umgebung [Roos02]“ 19
2.2.1.2: „Enhancement-Vorgang“ 21
2.2.4.1: „JDO-Transparenz [Roos02]“ 27
2.2.4.4: „Zustandsdiagramm der nach JDO-Spezifikation geforderten sieben Zustände“ 31
2.2.7: „Die Exception-Hierarchie [Roos02]“ 37
3.2: „Klassendiagramm des Prototyp“ 44
3.2.1.1: „Entity-Relationship-Modell der Datenbank diplom“ 47
3.2.1.2: „Persistenz-Deskriptor der Klasse Primary“ 49
3.2.2.2: „Entity-Relationship-Modell der Datenbank hibernate“ 54
3.2.3.2: „Entity-Relationship-Modell der Datenbank castordiplom“ 60
3.2.4.1: „Entity-Relationship-Modell der Datenbank kodoDiplom“ 65
3.2.5.1: “Entity-Relationship-Modell der Datenbank lidodiplom” 71
4.1.3: „Diagramm Serialisierung, Deserialisierung und Speicherverbrauch“ 76
4.2.4.6: „Performance der DB-Operationen unter HSQL im Server-Modus“ 91
4.2.4.6: „Performance der DB-Operationen unter HSQL im embedded-Modus“ 91
4.3.3.6: „Performance-Ergebnisse unter Verwendung der Datenbank HSQL im eingebetteten Modus“ 98
4.4.5.6: „Performance-Ergebnisse unter Verwendung der Datenbank HSQL im Server-Modus“ 106
4.4.5.6: „Performance-Ergebnisse unter Verwendung der Datenbank HSQL im embedded-Modus“ 106
4.5.3.6: „Performance-Ergebnisse unter Verwendung der Datenbank HSQL im Server-Modus“ 113
4.5.3.6: „Performance-Ergebnisse unter Verwendung der Datenbank HSQL im embedded-Modus“ 113
6.6: „Persistenz-Deskriptor für das Objektmodell unter Verwendung von Hibernate“ 126
6.7: „SQL-Skript für die Datenbank hibernate“ 127
6.8: „Persistenz-Deskriptor für das Objektmodell unter Verwendung von Castor JDO“ 130
6.9: „SQL-Skript für die Datenbank castordiplom“ 131
6.10: „Persistenz-Deskriptor für das Objektmodell unter Verwendung von Kodo JDO“ 133
6.11: „SQL-Skript für die Datenbank kododiplom“ 134
6.12: „Persistenz-Deskriptor für das Objektmodell unter Verwendung von Lido“ 135
6.13: „SQL-Skript für die Datenbank lidodiplom“ 137
6.14: „DTD-File jdo.dtd von der JDO-Spezifikation“ 138

6.2 JDO-Implementierungen-Quellen

- FastObjects (http://www.fastobjects.com/)
- JRELAY V2.0 (http://www.objectindustries.com/)
- KODO JDO (http://www.solarmetric.com/Software/Kodo_JDO/)
- IntelliBO V3.1 (http://www.signsoft.de)
- LIDO (http://www.libelis.com)
- OpenFusion (http://www.prismtechnologies.com)
- Orient (http://www.orientechnologies.com)
- JDO- Genie (http://www.hemtech.co.za/jdo/index.html)
- JORM (http://www.objectweb.org/jorm/index.html)
- QBeans JDO (http://www.qtech.ca/qbeansJDO.html)
- RexIP JDO (http://www.rexip.com/products/jdo/index.jsp)
- FrontierSuite JDO (http://www.objectfrontier.com/jdo)
- EnJin bzw. Judo (http://www.versant.com)
- Castor JDO (http://castor.exolab.org/jdo.html)
- TJDO (http://tjdo.sourceforge.net)
- XORM (http://xorm.sourceforge.net)

6.3 Ergebnisse der Internetrecherche

Es folgt eine Tabelle, in der die einzelnen JDO-Implementierungs-Features aufgelistet sind:

	FastObjects e7 Poet	**Jrelay V2.0 object industries**	**KODO JDO V2.4.3 Solarmetric**	**IntelliBO V3.1 Signsoft**
Lizenzart	Kommerziell	Kommerziell	Kommerziell	Kommerziell
Kosten	Auf Anfrage	Professional: 999$ Enterprise: 1999$	Standard: 600$ Enterprise: 3000$	Professional: 2450Euro
JDO-Standard-konform bzw. API	Ja	Ja	Ja	Ja
MySQL-Unterstützung	Ja	Wird nicht explizit erwähnt	Ja	Ja
Optionale Datentypen	Alle	Alle	Alle	Alle
Transaktionskonzept	Beide	Beide	Beide	Beide
GUI-Entwicklungsumgebung	Plug-Ins	ja	Plug-Ins	Ja
Enhancement-Werkzeug	ja	Ja	Ja	Ja
Tabellenskript-Generator	ja	Ja	Ja	Ja
EJB-Unterstützung	Nein, die Version t7 schon	Nur Enterprise	Nur Enterprise	Ja
Unterstützung JDOQL	Ja	Ja	Ja	Ja
Identitätskonzept	Alle	Alle	Alle	Alle

	LIDO V1.4 Libelis	**Orient Orient Technologies**	**JDO-Genie Hemisphere Technologies**	**XORM**
Lizenzart	Kommerziell	Kommerziell	Kommerziell	Open-source
Kosten	Standard: 600Euro Professional: 2000Euro	Just-Edition: 199Euro	Professional: 500$ Enterprise: 2000$	Frei
JDO-Standard-konform bzw. API	Ja	Ja	Ja	Ja
MySQL-Unterstützung	Ja	Eigene ODBMS	Ja	Ja
Optionale Datentypen	Alle	Fast alle	Fast alle	Teils
Transaktionskonzept	Beide	?	Beide	Nur pess.
GUI-Entwicklungsumgebung	Nur Professional	Ja	Ja	Nein
Enhancement-Werkzeug	Ja	Nein	Ja	Nein
Tabellenskript-Generator	Ja	Ja	Ja	Ja
EJB-Unterstützung	Nur Professional	Nein	Nur Enterprise	Nein
Unterstützung JDOQL	Ja	Ja	Ja	Ja
Identitätskonzept	Alle	?	Alle	Nur DB-Ident

	Qbeans JDO	RexIP JDO (TradeCity CyberSoft)	Castor JDO (exolab)	TJDO
Lizenzart	Kommerziell	Kommerziell	Open-source	Open-source
Kosten	Auf Anfrage	Auf Anfrage	Frei	Frei
JDO-Standard-konform bzw. API	Ja	Ja	Nein	Ja
MySQL-Unterstützung	Ja	Ja	Ja	Ja
Optionale Datentypen	Ja	ja	Teils	Nein
Transaktionskonzept	Beide	Beide	Beide	Nur pess.
GUI-Entwicklungsumgebung	Nein	Plug-Ins	Nein	Nein
Enhancement-Werkzeug	Ja	Ja	Nein	Ja
Tabellenskript-Generator	Ja	Ja	Nein	Ja
EJB-Unterstützung	Ja	Ja	Ja	Nein
Unterstützung JDOQL	Ja	Ja	Nein	Ja
Identitätskonzept	Alle	Alle	Eigenes	Nur DB-Ident

6.4 Installation und Konfiguration von MySQL

MySQL benutzt das Netzwerkprotokoll TCP/IP um die Verbindung von einem Client zu einem Server herzustellen. Bevor MySQL installiert bzw. gestartet wird, muss TCP/IP installiert sein.

1. Unter http://www.mysql.com/downloads/index.html die aktuellste MySQL-Version herunterladen (in unserem Falle mysql-3.23.54-win.zip)

2. Unzip mysql-3.23.54-win.zip und die extrahierte Setup.exe ausführen, MySQL wird installiert, und es wird automatisch eine Test-Datenbank „test“ angelegt

3. Unter http://www.mysql.com/downloads/gui-mysqlgui.html das zugehörige GUI-Tool herunterladen, entpacken und installieren. Dies ermöglicht eine einfache Verwaltung von MySQL-Datenbanken. Mit Hilfe dieses Tool können direkt SQL-Statements auf der Datenbank ausgeführt werden.

4. Des weiteren unter http://www.mysql.com/downloads/api-jdbc.html den MM-JDBC-Treiber oder den offiziellen JDBC-Treiber Connector/J http://www.mysql.com/downloads/api-jdbc-stable.html für MySQL herunterladen und entpacken

5. Im Standardinstallationsverzeichnis c:\mysql befindet sich im Directory bin der MySQL Serverdämon „`mysqld-max-nt.exe`“ und der MySQL-AdminManager „`winmysqladmin.exe`“.

6. Mit Hilfe eines Editors muss man jetzt noch die Datei „`my.ini`“ im Windows–Verzeichnis anpassen

Wenn man den Tabellentyp innoDB verwendet, muss man „von Hand“ zwei neue Verzeichnisse anlegen (c:\ibdata und c:\iblogs), die die innoDB Daten und die dazugehörigen Logfiles beinhalten.

Auszug my.ini:

```
[mysqld]
basedir=C:/mysql
datadir=C:/mysql/data
```

```
#innoDB- Konfigurationen Verzeichnisse müssen vorher von Hand angelegt
#werden
innodb_data_home_dir = c:\ibdata
innodb_log_group_home_dir = c:\iblogs
innodb_log_arch_dir = c:\iblogs

#zu Beginn wird ein einzelnes File mit einer Grösse von 50MB angelegt, und durch
#:autoextend automatisch bei vollem Speicher in 8MB Schritten vergrößert wird
innodb_data_file_path = ipdata1:50M

set-variable = innodb_buffer_pool_size=100M
set-variable = innodb_additional_mem_pool_size=20M
set-variable = innodb_log_file_size=10M
set-variable = innodb_log_buffer_size=5M

# kann auf 0 gesetzt werden, wenn man sich leisten kann, einige der zuletzt abgeschickten
Transaktionen
# (commit) zu verlieren.
innodb_flush_log_at_trx_commit=1
set-variable = innodb_file_io_threads=4
set-variable = innodb_lock_wait_timeout=50
```

7. Durch Aufruf von „`mysqld-max-nt --install`“ installiert man den MySQL-Server als Dienst/Service unter NT/Windows2000, d.h. der MySQL-Server läuft permanent im Hintergrund, der Datenbankanfragen entgegen nehmen kann. Als Service bedeutet, dass das Betriebssystem den Server automatisch stoppt, wenn der Rechner herunter gefahren wird. Es werden folgende Tabellentypen unterstützt: MyISAM (Standard-MySQL-Tabellentyp), HEAP, ISAM (deprecated) und MERGE, jeweils ohne Transaktionsunterstützung; InnoDB und BDB (beide Tabellentypen unterstützen Transaktionen).

8. Durch Aufruf von „mysql“ startet man den MySQL-Monitor. Nun muss durch den Befehl „create database diplom“ die Datenbank diplom angelegt, durch „use diplom“ diese als aktive Datenbank festgelegt werden und anschließend durch den Befehl „\. C:\prototype\model\Schemas\create.sql“ die Tabellen angelegt werden.

9. Durch Aufruf von „mysqladmin shutdown“ stoppt man schließlich den MySQL-Server

10. Errormeldungen werden in der Datei mysql.err im Installationsverzeichnis unter „`c:\mysql\data\mysql.err`“ `protokolliert`

6.5 Installation und Konfiguration von Hypersonic SQL (HSQL)

1. Unter http://hsqldb.sourceforge.net/ die aktuellste HSQL- Version herunterladen (hier: hsql_1_7_1.zip)

2. Unzip hsql_1_7_1.zip . Es wird unter anderem ein Verzeichnis hsqldb\lib angelegt, indem sich das JAR-File `hsqldb.jar` befindet.

3. Dieses JAR-File muss in den CLASSPATH aufgenommen werden. Innerhalb des JAR-File befindet sich auch der JDBC-Treiber `org.hsqldb.jdbcDriver`

4. Durch Aufruf von `java org.hsqldb.util.DatabaseManager` wird der mitgelieferte graphische Datenbank-Manager gestartet. Zu Beginn muss entschieden werden, ob die Verwaltung (Erzeugung der Datenbank hsqldiplom und die zugehörigen Tabellen) für die eingebette (HSQL Database Engine Standalone) oder für den Server-Modus (HSQL Database Engine Server) erfolgen soll. Als JDBC-Treiber wird `org.hsqldb.jdbcDriver` angegeben.

5. Für den eingebetteten Modus muss die URL noch mit dem Datenbanknamen versehen werden: jdbc:hsqldb:«database?» → jdbc:hsqldb:castorHSQLdiplom. Für den Server-Modus lautet die URL: jdbc:hsqldb:hsql://localhost

6. Über den Menupunkt File/OpenSkript muss das sich im Verzeichnis c:\castorHSQL\Schemas\create-HSQL.sql befindliche SQL-Skript ausgeführt werden. Es wird eine Datei namens castorHSQLdiplom.script erzeugt, die die Daten der Tabellen verwaltet. Des weiteren wird noch eine Datei namens castorHSQLdiplom.properties erzeugt, in der z.B. Cache-Einstellungen vorgenommen werden können. Es wurden die Default-Einstellungen übernommen.

7. Per Default-Einstellung befindet sich HSQL im AutoCommit-Modus. Dieser muss über den Menupunkt Options/AutoCommit-Off ausgeschaltet werden.

8. Soll die Verwaltung über den DatabaseManager für den Server-Modus erfolgen, so muss der HSQL-Server zuvor durch Aufruf von: java org.hsqldb.Server gestartet worden sein.

6.6 Persistenz-Deskriptor Hibernate

Es folgt der für das Objektmodell erstellte Persistenz-Deskriptor „`prototype.hbm.xml`“:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
“http://hibernate.sourceforge.net/hibernate-mapping-1.1.dtd">
<hibernate-mapping>

    <class name="hibernate.AnotherClass" table="tanotherclass">
        <id name="intValueAC" type="int" column="intValueAC">
            <generator class="assigned"/>
        </id>
        <many-to-one name="secondarySecond" column="secondarySecond_ID"
            class="hibernate.SecondarySecond" cascade="delete" />
        <property name="stringValueAC" column="stringValueAC" type="string"/>
    </class>

    <class name="hibernate.BasePrimary" table="tbaseprimary">
        <id name="intValueBP" type="int" column="intValueBP">
            <generator class="assigned"/>
        </id>
        <property name="stringValueBP" column="stringValueBP" type="string"/>
        <property name="boolValueBP" column="boolValueBP" type="boolean"/>
        <list role="arrayListBP" table="tprimarysecond" lazy="false" cascade="delete">
            <key column="basePrimary_ID"/>
            <index column="posn" type="int"/>
            <one-to-many class="hibernate.PrimarySecond"/>
        </list>
```

```
    <joined-subclass name="hibernate.Primary" table="tprimary">
        <key column="intValueBP"/>
        <property name="stringValueP" column="stringValueP" type="string"/>
        <property name="boolValueP" column="boolValueP" type="boolean"/>
        <list role="arrayListP" table="tsecondary" lazy="false" cascade="delete">
            <key column="primary_ID_list"/>
            <index column="posn" type="int"/>
            <one-to-many class="hibernate.Secondary"/>
        </list>
        <map role="propertiesP" table="tpropertiesp" lazy="false" cascade="delete">
            <key column="primary_ID_prop"/>
            <index column="keyp" type="string"/>
            <element column="valuep" type="string"/>
        </map>
        <map role="hashMapP" table="tsecondary" lazy="false" cascade="delete">
            <key column="primary_ID"/>
            <index column="hashMapPKey" type="int"/>
            <one-to-many class="hibernate.Secondary"/>
        </map>
    </joined-subclass>
</class>

<class name="hibernate.PrimaryFirst" table="tprimaryfirst">
    <id name="intValuePF" type="int" column="intValuePF">
        <generator class="assigned"/>
    </id>
    <property name="boolValuePF" column="boolValuePF" type="boolean"/>
    <property name="stringValuePF" column="stringValuePF" type="string"/>
</class>

<class name="hibernate.PrimarySecond" table="tprimarysecond">
    <id name="intValuePS" type="int" column="intValuePS">
        <generator class="assigned"/>
    </id>
    <many-to-one name="primaryFirst" column="primaryFirst_ID" class="hibernate.PrimaryFirst"
        cascade="delete" />
    <property name="stringValuePS" column="stringValuePS" type="string"/>
</class>

<class name="hibernate.BaseSecondary" table="tbasesecondary">
    <id name="intValueBS" type="int" column="intValueBS">
        <generator class="assigned"/>
    </id>
    <property name="doubleValueBS" column="doubleValueBS" type="double"/>
    <property name="stringValueBS" column="stringValueBS" type="string"/>
    <many-to-one name="secondaryFirst" column="secondaryFirst_ID" class="hibernate.SecondaryFirst"
        cascade="all" />
    <list role="arrayListBS" table="tsecondarysecond" lazy="false" cascade="delete" >
        <key column="baseSecondary_ID"/>
        <index column="posn" type="int"/>
        <one-to-many class="hibernate.SecondarySecond"/>
    </list>

    <joined-subclass name="hibernate.Secondary" table="tsecondary">
        <key column="intValueBS"/>
        <property name="boolValueS" column="boolValueS" type="boolean"/>
        <property name="stringValueS" column="stringValueS" type="string"/>
        <list role="arrayListS" table="tanotherclass" lazy="false" cascade="delete">
            <key column="secondary_ID_list"/>
            <index column="posn" type="int"/>
            <one-to-many class="hibernate.AnotherClass"/>
        </list>
        <map role="hashMapS" table="tanotherclass" lazy="false" cascade="delete">
            <key column="secondary_ID"/>
            <index column="hashMapSKey" type="int"/>
            <one-to-many class="hibernate.AnotherClass"/>
        </map>
    </joined-subclass>
</class>
```

```
    <class name="hibernate.SecondaryFirst" table="tsecondaryfirst">
        <id name="intValueSF" type="int" column="intValueSF">
            <generator class="assigned"/>
        </id>
        <property name="stringValueSF" column="stringValueSF" type="string"/>
        <property name="boolValueSF" column="boolValueSF" type="boolean"/>
    </class>

    <class name="hibernate.SecondarySecond" table="tsecondarysecond">
        <id name="intValueSS" type="int" column="intValueSS">
            <generator class="assigned"/>
        </id>
        <property name="stringValueSS" column="stringValueSS" type="string"/>
    </class>

</hibernate-mapping>
```

Abbildung 21„Persistenz-Deskriptor für das Objektmodell unter Verwendung von Hibernate“

6.7 SQL-Skript für die Datenbank hibernate

```
create table tpropertiesp
(
        primary_ID_prop int(10) not null,
        valuep varchar(50),
        keyp varchar(50) not null,
        primary key (primary_ID_prop, keyp)
) type=innoDB;

alter table tpropertiesp add index (primary_ID_prop),
add constraint fk_prop_P foreign key (primary_ID_prop)
references tprimary (intValueBP);
```

```
create table tbasesecondary
(
        intValueBS int(10) not null,
        doubleValueBS double,
        stringValueBS varchar(50),
        secondaryFirst_ID int(10),
        primary key (intValueBS)
) type=innoDB;

alter table tbasesecondary add index (secondaryFirst_ID),
add constraint fk_BS_SF foreign key (secondaryFirst_ID)
references tsecondaryfirst (intValueSF);
```

```
create table tprimarysecond
(
        intValuePS int(10) not null,
        primaryFirst_ID int(10),
        stringValuePS varchar(50),
        basePrimary_ID int(10),
        posn int(10),
        primary key (intValuePS)
) type=innoDB;

alter table tprimarysecond add index (basePrimary_ID),
```

```
create table tsecondarysecond
(
        intValueSS int(10) not null,
        stringValueSS varchar(50),
        baseSecondary_ID int(10),
        posn int(10),
        primary key (intValueSS)
) type=innoDB;

alter table tsecondarysecond add index (baseSecondary_ID),
add constraint fk_SS_BS foreign key (baseSecondary_ID)
```

add constraint **fk_PS_BP** foreign key (basePrimary_ID) references tbaseprimary (intValueBP); alter table tprimarysecond add index (primaryFirst_ID), add constraint **fk_PS_PF** foreign key (primaryFirst_ID) references tprimaryfirst (intValuePF);	references tbasesecondary (intValueBS);
create table **tprimaryfirst** (intValuePF int(10) not null, boolValuePF bit, stringValuePF varchar(50), primary key (intValuePF)) type=innoDB;	create table **tsecondaryfirst** (intValueSF int(10) not null, stringValueSF varchar(50), boolValueSF bit, primary key (intValueSF)) type=innoDB;
create table **tbaseprimary** (intValueBP int(10) not null, stringValueBP varchar(50), boolValueBP bit, primary key (intValueBP)) type=innoDB;	create table **tprimary** (intValueBP int(10) not null, stringValueP varchar(50), boolValueP bit, primary key (intValueBP)) type=innoDB; alter table tprimary add index (intValueBP), add constraint **fk_P_BP** foreign key (intValueBP) references tbaseprimary (intValueBP);
create table **tanotherclass** (intValueAC int(10) not null, secondarySecond_ID int(10), stringValueAC varchar(50), secondary_ID int(10), hashMapSKey int(10), secondary_ID_list int(10), posn int(10), primary key (intValueAC)) type=innoDB; alter table tanotherclass add index (secondarySecond_ID), add constraint **fk_AC_SS** foreign key (secondarySecond_ID) references tsecondarysecond (intValueSS); alter table tanotherclass add index (secondary_ID), add constraint **fk_AC_S_map** foreign key (secondary_ID) references tsecondary (intValueBS); alter table tanotherclass add index (secondary_ID_list), add constraint **fk_AC_S_list** foreign key (secondary_ID_list) references tsecondary (intValueBS);	create table **tsecondary** (intValueBS int(10) not null, boolValueS bit, stringValueS varchar(50), primary_ID_list int(10), posn int(10), primary_ID int(10), hashMapPKey int(10), primary key (intValueBS)) type=innoDB; alter table tsecondary add index (primary_ID_list), add constraint **fk_S_P** foreign key (primary_ID_list) references tprimary (intValueBP); alter table tsecondary add index (intValueBS), add constraint **fk_S_BS** foreign key (intValueBS) references tbasesecondary (intValueBS); alter table tsecondary add index (primary_ID), add constraint **fk_S_P** foreign key (primary_ID) references tprimary (intValueBP);

Abbildung 22„SQL-Skript für die Datenbank hibernate“

6.8 Persistenz-Deskriptor Castor JDO

Es folgt der für das Objektmodell erstellte Persistenz-Deskriptor „`mapping.xml`“:

```
<!DOCTYPE databases PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
"http://castor.exolab.org/mapping.dtd">

<mapping>
    <class name="castor.PrimarySecond" identity="intValuePS" depends="BasePrimary">
        <map-to table="tprimarysecond"/>
            <cache-type type="unlimited"/>
        <field name="intValuePS" type="integer" required="true">
            <sql name="intValuePS" type="integer"/>
        </field>
        <field name="stringValuePS" type="string">
            <sql name="stringValuePS" type="varchar"/>
        </field>
        <field name="primaryFirst" type="castor.PrimaryFirst">
            <sql name="primaryFirst_ID"/>
        </field>
        <field name="bp" type="castor.BasePrimary">
            <sql name="basePrimary_ID"/>
        </field>
```

```
</class>

<class name="castor.PrimaryFirst" identity="intValuePF">
    <map-to table="tprimaryfirst"/>
        <cache-type type="unlimited"/>
    <field name="intValuePF" type="integer" required="true">
        <sql name="intValuePF" type="integer"/>
    </field>
    <field name="boolValuePF" type="boolean">
        <sql name="boolValuePF" type="bit"/>
    </field>
    <field name="stringValuePF" type="string">
        <sql name="stringValuePF" type="varchar"/>
    </field>
</class>

<class name="castor.BasePrimary" identity="intValueBP">
    <map-to table="tbaseprimary"/>
        <cache-type type="unlimited"/>
    <field name="intValueBP" type="integer" required="true">
        <sql name="intValueBP" type="integer"/>
    </field>
    <field name="stringValueBP" type="string">
        <sql name="stringValueBP" type="varchar"/>
    </field>
    <field name="boolValueBP" type="boolean">
        <sql name="boolValueBP" type="bit"/>
    </field>
    <field name="arrayListBP" type="castor.PrimarySecond" collection="arraylist">
        <sql many-key="basePrimary_ID"/>
    </field>
</class>

<class name="castor.BaseSecondary" identity="intValueBS">
    <map-to table="tbasesecondary"/>
        <cache-type type="unlimited"/>
    <field name="intValueBS" type="integer" required="true">
        <sql name="intValueBS" type="integer"/>
    </field>
    <field name="doubleValueBS" type="double">
        <sql name="doubleValueBS" type="double"/>
    </field>
    <field name="stringValueBS" type="string">
        <sql name="stringValueBS" type="varchar"/>
    </field>
    <field name="secondaryFirst" type="castor.SecondaryFirst">
        <sql name="secondaryFirst_ID"/>
    </field>
    <field name="arrayListBS" type="castor.SecondarySecond" collection="arraylist">
        <sql many-key="baseSecondary_ID"/>
    </field>
</class>

<class name="castor.AnotherClass" identity="intValueAC" depends="Secondary">
    <map-to table="tanotherclass"/>
        <cache-type type="unlimited"/>
    <field name="intValueAC" type="integer" required="true">
        <sql name="intValueAC" type="integer"/>
    </field>
    <field name="stringValueAC" type="string">
        <sql name="stringValueAC" type="varchar"/>
    </field>
    <field name="secondarySecond" type="castor.SecondarySecond">
        <sql name="secondarySecond_ID"/>
    </field>
    <field name="secList" type="castor.Secondary">
        <sql name="secondary_ID_list"/>
    </field>
</class>

<class name="castor.SecondarySecond" identity="intValueSS" depends="BaseSecondary">
```

```
    <map-to table="tsecondarysecond"/>
        <cache-type type="unlimited"/>
    <field name="intValueSS" type="integer" required="true">
        <sql name="intValueSS" type="integer"/>
    </field>
    <field name="stringValueSS" type="string">
        <sql name="stringValueSS" type="varchar"/>
    </field>
    <field name="bs" type="castor.BaseSecondary">
        <sql name="baseSecondary_ID"/>
    </field>
</class>

<class name="castor.SecondaryFirst" identity="intValueSF">
    <map-to table="tsecondaryfirst"/>
        <cache-type type="unlimited"/>
    <field name="intValueSF" type="integer" required="true">
        <sql name="intValueSF" type="integer"/>
    </field>
    <field name="stringValueSF" type="string">
        <sql name="stringValueSF" type="varchar"/>
    </field>
    <field name="boolValueSF" type="boolean">
        <sql name="boolValueSF" type="bit"/>
    </field>
</class>

<class name="castor.Primary" extends="castor.BasePrimary" identity="intValueBP">
    <map-to table="tprimary"/>
        <cache-type type="unlimited"/>
    <field name="intValueBP" type="integer" required="true">
        <sql name="intValueBP" type="integer"/>
    </field>
    <field name="stringValueP" type="string">
        <sql name="stringValueP" type="varchar"/>
    </field>
    <field name="boolValueP" type="boolean">
        <sql name="boolValueP" type="bit"/>
    </field>
    <field name="arrayListP" type="castor.Secondary" collection="arraylist">
        <sql many-key="primary_ID_list"/>
    </field>
    <field name="hashMapP" type="castor.Secondary" collection="map">
        <sql many-key="primary_ID_list"/>
    </field>
    <field name="propertiesP" type="castor.Property" collection="collection">
        <sql many-key=”primary_ID_prop”/>
    </field>
</class>

<class name="castor.Secondary" extends="castor.BaseSecondary" identity="intValueBS"
        depends="Primary">
    <map-to table="tsecondary"/>
        <cache-type type="unlimited"/>
    <field name="intValueBS" type="integer" required="true">
        <sql name="intValueBS" type="integer"/>
    </field>
    <field name="boolValueS" type="boolean">
        <sql name="boolValueS" type="bit"/>
    </field>
    <field name="stringValueS" type="string">
        <sql name="stringValueS" type="varchar"/>
    </field>
    <field name="arrayListS" type="castor.AnotherClass" collection="arraylist">
        <sql many-key="secondary_ID_list"/>
    </field>
    <field name="hashMapS" type="castor.AnotherClass" collection="map">
        <sql many-key="secondary_ID_list“
    </field>
    <field name="primList" type="castor.Primary" >
```

```
            <sql name="primary_ID_list"/>
        </field>
    </class>

    <class name="castor.Property" identity="key" depends="Primary">
        <map-to table="tpropertiesp"/>
            <cache-type type="unlimited"/>
        <field name="key" type="string" required="true">
            <sql name="keyp" type="varchar" />
        </field>
        <field name="value" type="string" required="true">
            <sql name="valuep" type="varchar"/>
        </field>
    </class>

</mapping>
```

Abbildung 23 „Persistenz-Deskriptor für das Objektmodell unter Verwendung von Castor JDO“

6.9 SQL-Skript für die Datenbank castordiplom

```
create table tprimarysecond
(
        intValuePS int(10) not null,
        basePrimary_ID int(10),
        stringValuePS varchar(50),
        primaryFirst_ID int(10),
        primary key (intValuePS)
)
type=innoDB;
```

```
create table tprimaryfirst
(
        intValuePF int(10) not null,
        stringValuePF varchar(50),
        boolValuePF bit,
        primary key (intValuePF)
)
type=innoDB;
```

```
create table tbaseprimary
(
        intValueBP int(10) not null,
        stringValueBP varchar(50),
        boolValueBP bit,
        primary key (intValueBP)
)
type=innoDB;
```

```
create table tprimary
(
        intValueBP int(10) not null,
        boolValueP bit,
        stringValueP varchar(50),
        primary key (intValueBP)
)
type=innoDB;
```

```
create table tanotherclass
(
        intValueAC int(10) not null,
        stringValueAC varchar(50),
        secondarySecond_ID int(10),
        secondary_ID_list int(10),
        primary key (intValueAC)
)
type=innoDB;
```

```
create table tsecondaryfirst
(
        intValueSF int(10) not null,
        stringValueSF varchar(50),
        boolValueSF bit,
        primary key (intValueSF)
)
type=innoDB;
```

```
create table tbasesecondary
(
        intValueBS int(10) not null,
        stringValueBS varchar(50),
        doubleValueBS double,
        secondaryFirst_ID int(10),
```

```
create table tsecondarysecond
(
        intValueSS int(10) not null,
        stringValueSS varchar(50),
        baseSecondary_ID int(10),
        primary key (intValueSS)
```

```
        primary key (intValueBS)
)
type=innoDB;
```

```
)
type=innoDB;
```

```
create table tsecondary
(
        intValueBS int(10) not null,
        stringValueS varchar(50),
        boolValueS bit,
        primary_ID_list int(10),
        primary key (intValueBS)
)
type=innoDB;
```

```
create table tpropertiesp
(
        primary_ID_prop integer(10) not null,
        keyp varchar(50) not null,
        valuep varchar(50) not null,
        primary key (primary_ID_prop)

)
type=innoDB;
```

Abbildung 24„SQL-Skript für die Datenbank castordiplom“

6.10Persistenz-Deskriptor Kodo JDO

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
<package name="kododiplom">

<class identity-type="application" name="AnotherClass" objectid-class="AnotherClassKey" requires-
extent="true">
    <extension vendor-name="kodo" key="table" value="tanotherclass"/>
    <extension vendor-name="kodo" key="lock-column" value="none"/>
    <extension vendor-name="kodo" key="class-column" value="none"/>
    <field name="secondarySecond">
        <extension vendor-name="kodo" key="dependent" value="true"/>
    </field>
    <field name="intValueAC" primary-key="true"/>
</class>

<class identity-type="application" name="BasePrimary" objectid-class="BasePrimaryKey" requires-extent="true">
    <extension vendor-name="kodo" key="table" value="tbaseprimary"/>
    <extension vendor-name="kodo" key="lock-column" value="none"/>
    <field name="intValueBP" primary-key="true"/>
    <field name="arrayListBP">
        <collection element-type="kododiplom.PrimarySecond"/>
        <extension vendor-name="kodo" key="element-dependent" value="true"/>
        <extension vendor-name="kodo" key="inverse" value="bp"/>
    </field>
</class>

<class identity-type="application" name="BaseSecondary" objectid-class="BaseSecondaryKey" requires-
extent="true">
    <extension vendor-name="kodo" key="table" value="tbasesecondary"/>
    <extension vendor-name="kodo" key="lock-column" value="none"/>
    <field name="intValueBS" primary-key="true"/>
    <field name="arrayListBS">
        <collection element-type="kododiplom.SecondarySecond"/>
        <extension vendor-name="kodo" key="element-dependent" value="true"/>
```

```
        <extension vendor-name="kodo" key="inverse" value="bs"/>
    </field>
    <field name="secondaryFirst">
        <extension key="dependent" value="true" vendor-name="kodo"/>
    </field>
</class>

<class identity-type="application" name="Primary" persistence-capable-superclass="BasePrimary" requires-extent="true">
    <extension vendor-name="kodo" key="table" value="tprimary"/>
    <extension vendor-name="kodo" key="lock-column" value="none"/>
    <extension vendor-name="kodo" key="class-column" value="none"/>
    <field name="hashMapP">
        <map key-type="java.lang.Integer" value-type="kododiplom.Secondary"/>
        <extension vendor-name="kodo" key="value-dependent" value="true"/>
            <extension vendor-name="kodo" key="key-dependent" value="true"/>
            <extension vendor-name="kodo" key="inverse" value="primMap"/>
    </field>
    <field name="arrayListP">
        <collection element-type="kododiplom.Secondary"/>
            <extension vendor-name="kodo" key="inverse" value="primList"/>
            <extension vendor-name="kodo" key="element-dependent" value="true"/>
    </field>
    <field name="propertiesP" persistence-modifier="persistent">
        <map key-type="java.lang.String" value-type="java.lang.String"/>
        <extension vendor-name="kodo" key="value-dependent" value="true"/>
        <extension vendor-name="kodo" key="key-dependent" value="true"/>
    </field>
</class>

<class identity-type="application" name="PrimarySecond" objectid-class="PrimarySecondKey" requires-extent="true">
    <extension vendor-name="kodo" key="table" value="tprimarysecond"/>
    <extension vendor-name="kodo" key="lock-column" value="none"/>
    <extension vendor-name="kodo" key="class-column" value="none"/>
    <field name="intValuePS" primary-key="true"/>
    <field name="primaryFirst">
        <extension vendor-name="kodo" key="dependent" value="true"/>
      </field>
</class>

<class identity-type="application" name="PrimaryFirst" objectid-class="PrimaryFirstKey" requires-extent="true">
    <extension vendor-name="kodo" key="table" value="tprimaryfirst"/>
    <extension vendor-name="kodo" key="lock-column" value="none"/>
    <extension vendor-name="kodo" key="class-column" value="none"/>
    <field name="intValuePF" primary-key="true"/>
</class>

<class identity-type="application" name="Secondary" persistence-capable-superclass="BaseSecondary" requires-extent="true">
    <extension vendor-name="kodo" key="table" value="tsecondary"/>
    <extension vendor-name="kodo" key="lock-column" value="none"/>
    <extension vendor-name="kodo" key="class-column" value="none"/>
    <field name="hashMapS">
        <map key-type="java.lang.Integer" value-type="kododiplom.AnotherClass"/>
            <extension vendor-name="kodo" key="value-dependent" value="true"/>
            <extension vendor-name="kodo" key="key-dependent" value="true"/>
            <extension vendor-name="kodo" key="inverse" value="secMap"/>
    </field>
    <field name="arrayListS">
        <collection element-type="kododiplom.AnotherClass"/>
            <extension vendor-name="kodo" key="inverse" value="secList"/>
            <extension vendor-name="kodo" key="element-dependent" value="true"/>
    </field>
</class>

<class identity-type="application" name="SecondaryFirst" objectid-class="SecondaryFirstKey" requires-extent="true">
    <extension vendor-name="kodo" key="table" value="tsecondaryfirst"/>
    <extension vendor-name="kodo" key="lock-column" value="none"/>
    <extension vendor-name="kodo" key="class-column" value="none"/>
```

```
    <field name="intValueSF" primary-key="true"/>
</class>

<class identity-type="application" name="SecondarySecond" objectid-class="SecondarySecondKey" requires-
extent="true">
    <extension vendor-name="kodo" key="table" value="tsecondarysecond"/>
    <extension vendor-name="kodo" key="lock-column" value="none"/>
    <extension vendor-name="kodo" key="class-column" value="none"/>
    <field name="intValueSS" primary-key="true"/>
</class>

</package>
</jdo>
```

Abbildung 25„Persistenz-Deskriptor für das Objektmodell unter Verwendung von Kodo JDO“

6.11SQL-Skript für die Datenbank kodoDiplom

```
create table tanotherclass
(
        intValueACx int(10) not null,
        intValueBS_seclistx int(10),
        intValueBS_secmapx int(10),
        intValueSS_secondarysecondx int(10),
        stringValueACx varchar(50),
        primary key (intValueACx)
)
type = innodb;

alter table tanotherclass add index (intValueBS_seclistx);
alter table tanotherclass add index
(intValueSS_secondarysecondx);
alter table tanotherclass add index (intValueBS_secmapx);
```

```
create table tprimarysecond
(
        intValueBP_bpx int(10),
        intValuePF_primaryfirstx int(10),
        intValuePSx int(10) not null,
        stringValuePSx varchar(50),
        primary key (intValuePSx)
)
type = innodb;

alter table tprimarysecond add index (intValueBP_bpx);
alter table tprimarysecond add index (intValuePF_primaryfirstx);
```

```
create table primary_hashmappx
(
        intValueBP_jdoidx int(10) not null,
        intValueBS_hashmappx int(10),
        jdokeyx int(10) not null,
        primary key (intValueBP_jdoidx, jdokeyx)
)
type = innodb;

alter table primary_hashmappx add index (intValueBP_jdoidx);
alter table primary_hashmappx add index
(intValueBS_hashmappx);
```

```
create table primary_propertiespx
(
        intValueBP_jdoidx int(10) not null,
        jdokeyx varchar(50) not null,
        propertiesPx varchar(50),
        primary key (intValueBP_jdoidx, jdokeyx)
)
type = innodb;

alter table primary_propertiespx add index (intValueBP_jdoidx);
```

```
create table tbaseprimary
(
        boolValueBPx smallint,
        intValueBPx int(10) not null,
        stringValueBPx varchar(50),
        jdoclassx varchar(50),
        primary key (intvalueBPx)
)
type = innodb;
```

```
create table tprimary
(
        boolValuePx smallint,
        intVluebPx int(10) not null,
        stringVlueP varchar(50),
        primary key (intValueBPx)
)
type = innodb;
```

```
create table tprimaryfirst
(
        boolValuePFx smallint,
        intValuePFx int(10) not null,
        stringValuePFx varchar(50),
        primary key (intValuePFx)
)
type = innodb;
```

```
create table tsecondarysecond
(
        intValueBS_bsx int(10),
        intValueSSx int(10) not null,
        stringValueSSx varchar(50),
        primary key (intValueSSx)
)
type = innodb;

alter table tsecondarysecond add index (intValueBS_bsx);
```

```
create table tsecondaryfirst
(
        boolValueSFx smallint,
```

```
create table secondary_hashmapsx
(
        intValueAC_hashmapsx int(10),
```

```
        intValueSFx int(10) not null,
        stringValueSFx varchar(50),
        primary key (intValueSFx)
)
type = innodb;
```

```
        intValueBS_jdoidx int(10) not null,
        jdokeyx int(10) not null,
        primary key (intValueBS_jdoidx, jdokeyx)
)
type = innodb;

alter table secondary_hashmapsx add index (intValueAC_hashmapsx);
alter table secondary_hashmapsx add index (intValueBS_jdoidx);
```

```
create table tbasesecondary
(
        doubleValueBSx decimal(5,5),
        intValueBSx int(10) not null,
        intValueSF_secondaryfirstx int(10),
        stringValueBSx varchar(50),
        jdoclassx varchar(50),
        primary key (intValueBSx)
)
type = innodb;

alter table tbasesecondary add index (intValueSF_secondaryfirstx);
```

```
create table tsecondary
(
        boolValueSx smallint,
        intValueBP_primlistx int(10),
        intValueBP_primmapx int(10),
        intValueBSx int(10) not null,
        stringValueSx varchar(50),
        primary key (intValueBSx)
)
type = innodb;

alter table tsecondary add index (intValueBP_primlistx);
```

Abbildung 26„SQL-Skript für die Datenbank kododiplom“

6.12Persistenz-Deskriptor Lido

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
<package name="lidodiplom">

<class identity-type="application" name="AnotherClass" objectid-class="lidodiplom.AnotherClassKey" requires-extent="true">
    <extension vendor-name="libelis" key="sql-name" value="tanotherclass"/>
    <field name="intValueAC" primary-key="true"/>
    <field name="secondarySecond">
    </field>
</class>

<class identity-type="application" name="BasePrimary" objectid-class="lidodiplom.BasePrimaryKey" requires-extent="true">
    <field name="intValueBP" primary-key="true"/>
    <field name="arrayListBP">
        <collection element-type="lidodiplom.PrimarySecond"/>
        <extension vendor-name="libelis" key="collection-type" value="java.util.ArrayList"/>
        <extension vendor-name="libelis" key="sql-reverse" value="javaField:bp"/>
    </field>
</class>

<class identity-type="application" name="BaseSecondary" objectid-class="lidodiplom.BaseSecondaryKey" requires-extent="true">
    <field name="intValueBS" primary-key="true"/>
    <field name="arrayListBS">
        <collection element-type="lidodiplom.SecondarySecond"/>
        <extension vendor-name="libelis" key="collection-type" value="java.util.ArrayList"/>
        <extension vendor-name="libelis" key="sql-reverse" value="javaField:bs"/>
    </field>
    <field name="secondaryFirst">
    </field>
</class>


<class identity-type="application" name="Primary" persistence-capable-superclass="lidodiplom.BasePrimary" requires-extent="true">
    <extension vendor-name="libelis" key="sql-name" value="tprimary"/>

    <field name="hashMapP">
        <map key-type="java.lang.Integer" value-type="lidodiplom.Secondary"/>
        <extension vendor-name="libelis" key="map-type" value="java.util.HashMap"/>
```

```
        <extension vendor-name="libelis" key="sql-reverse" value="javaField:primMap"/>
        </field>
    <field name="arrayListP">
        <collection element-type="lidodiplom.Secondary"/>
        <extension vendor-name="libelis" key="collection-type" value="java.util.ArrayList"/>
        <extension vendor-name="libelis" key="sql-reverse" value="javaField:primList"/>
        </field>
        <field name="propertiesP" persistence-modifier="persistent">
            <map key-type="java.lang.String" value-type="java.lang.String"/>
        </field>
</class>

<class identity-type="application" name="PrimarySecond" objectid-class="lidodiplom.PrimarySecondKey"
requires-extent="true">
    <extension vendor-name="libelis" key="sql-name" value="tprimarysecond"/>

    <field name="intValuePS" primary-key="true"/>
    <field name="primaryFirst">
      </field>
</class>


<class identity-type="application" name="PrimaryFirst" objectid-class="lidodiplom.PrimaryFirstKey" requires-
extent="true">
    <extension vendor-name="libelis" key="sql-name" value="tprimaryfirst"/>
    <field name="intValuePF" primary-key="true"/>
</class>



<class identity-type="application" name="Secondary" persistence-capable-
superclass="lidodiplom.BaseSecondary" requires-extent="true">
    <extension vendor-name="libelis" key="sql-name" value="tsecondary"/>

    <field name="hashMapS">
        <map key-type="java.lang.Integer" value-type="lidodiplom.AnotherClass"/>
        <extension vendor-name="libelis" key="map-type" value="java.util.HashMap"/>
        <extension vendor-name="libelis" key="sql-reverse" value="javaField:secMap"/>
        </field>
        <field name="arrayListS">
        <collection element-type="lidodiplom.AnotherClass"/>
        <extension vendor-name="libelis" key="collection-type" value="java.util.ArrayList"/>
        <extension vendor-name="libelis" key="sql-reverse" value="javaField:secList"/>
        </field>
</class>


<class identity-type="application" name="SecondaryFirst" objectid-class="lidodiplom.SecondaryFirstKey"
requires-extent="true">
    <extension vendor-name="libelis" key="sql-name" value="tsecondaryfirst"/>

    <field name="intValueSF" primary-key="true"/>
</class>


<class identity-type="application" name="SecondarySecond" objectid-class="lidodiplom.SecondarySecondKey"
requires-extent="true">
    <extension vendor-name="libelis" key="sql-name" value="tsecondarysecond"/>

    <field name="intValueSS" primary-key="true"/>
</class>


</package>
</jdo>
```

Abbildung 27„Persistenz-Deskriptor für das Objektmodell unter Verwendung von Lido“

6.13SQL-Skript für die Datenbank lidodiplom

```
create table tsecondarysecond
(
        intValueSS int(10) not null,
        stringValueSS varchar(50),
        bs_intValueBS int,
        primary key (intValueSS)
)
type=innodb;

alter table add index tsecondarysecond (bs_intValueBS);
```

```
create table tprimaryfirst
(
        intValuePF int not null,
        boolValuePF char(1),
        stringValuePF varchar(50),
        primary key (intValuePF)
)
type=innodb;
```

```
create table tsecondaryfirst
(
        intValueSF int not null,
        boolValueSF char(1),
        stringValueSF varchar(50),
        primary key (intValueSF)
)
type=innodb;
```

```
create table tprimarysecond
(
        intValuePS int not null,
        stringValuePS varchar(50),
        primaryfirst_intValuePF int,
        bp_intValueBP int,
        primary key (intValuePS)
)
type=innodb;

alter table tprimarysecond add index (primaryfirst_intValuePF);
alter table tprimarysecond add index (bp_intValueBP);
```

```
create table tsecondary
(
        intValueBS int not null,
        doubleValueBS double,
        stringValueBS varchar(50),
        secondaryfirst_intValueSF int,
        boolValueS char(1),
        stringValueS varchar(50),
        primlist_intValueBP int,
        primmap_intValueBP int,
        primary key (intValueBS)
)
type=innodb;

alter table tsecondary add index (secondaryfirst_intValueSF);
alter table tsecondary add index (primmap_intValueBP);
alter table tsecondary add index (primlist_intValuebP);
```

```
create table tanotherclass
(
        intValueAC int not null,
        stringValueAC varchar(50),
        secondarysecond_intValueSS int,
        seclist_intValueBS int,
        secmap_intValueBS int,
        primary key (intValueAC)
)
type=innodb;

alter table tanotherclass add index (secmap_intvaluebs);
alter table tanotherclass add index (seclist_intvaluebs);
alter table tanotherclass add index
(secondarysecond_intValueSS);
```

```
create table tsecondary_hashmaps
(
        lidofk_intValueBS int not null,
        lidokey int,
        lidovalue_intValueAC int,
        primary key (lidofk_intValueBS, lidokey)
)
type=innodb;

alter table tsecondary_hashmaps add index (lidofk_intValueBS);
alter table tsecondary_hashmaps add index (lidokey);
```

```
create table tsecondary_arraylistbs
(
        lidofk_intValueBS int not null,
        lidovalue_intValueSS int,
        lidopos double,
        primary key(lidofk_intValueBS)

)
type=innodb;

alter table tsecondary_arraylistbs add index (lidofk_intValueBS);
alter table tsecondary_arraylistbs add index (lidopos);
```

```
create table tprimary
(
        intValueBP int not null,
        boolValueBP char(1),
```

```
create table tprimary_propertiesp
(
        lidofk_intValueBP int not null,
        lidokey varchar(50),
```

```
        stringValueBP varchar(50),
        boolValueP char(1),
        stringValueP varchar(50),
        primary key (intValueBP)
)
type=innodb;
```

```
        lidovalue varchar(50),
        primary key (lidofk_intValueBP, lidokey)
)
type=innodb;

alter table tprimary_propertiesp add index (lidofk_intvaluebp);
alter table tprimary_propertiesp add index (lidokey);
```

```
create table tprimary_arraylistbp
(
        lidofk_intValueBP int not null,
        lidovalue_intValuePS int,
        lidopos double,
        primary key (lidofk_intValueBP)
)
type=innodb;

alter table tprimary_arraylistbp add index (lidofk_intvaluebp);
alter table tprimary_arraylistbp add index (lidopos);

alter table tbasesecondary add index (intValueSF_secondaryfirstx);
```

```
create table tprimary_arraylistp
(
        lidofk_intValueBP int not null,
        lidovalue_intValueBS int,
        lidopos double,
        primary key (lidofk_intValueBP)
)
type=innodb;

alter table tprimary_arraylistp add index (lidofk_intvaluebp);
alter table tprimary_arraylistp add index (lidopos);
```

```
create table tprimary_hashmapp
(
        lidofk_intValueBP int not null,
        lidokey int,
        lidovalue_intValueBS int,
        primary key (lidofk_intValueBP, lidokey)
)
type=innodb;

alter table tprimary_hashmapp add index (lidofk_intvaluebp);
alter table tprimary_hashmapp add index (lidokey);
```

```
create table tsecondary_arraylistbs
(
        lidofk_intValueBS int not null,
        lidovalue_intValueSS int,
        lidopos double,
        primary key(lidofk_intValueBS)

)
type=innodb;

alter table tsecondary_arraylistbs add index (lidofk_intValueBS);
alter table tsecondary_arraylistbs add index (lidopos);
```

Abbildung 28„SQL-Skript für die Datenbank lidodiplom“

6.14DTD-File

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT jdo (package)+>

<!ELEMENT package ((class)+, (extension)*)>
<!ATTLIST package name CDATA #REQUIRED>

<!ELEMENT class ((field|extension)*)>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class identity-type (application|datastore|nondurable) "datastore">
<!ATTLIST class objectid-class CDATA #IMPLIED>
<!ATTLIST class requires-extent (true|false) "true">
<!ATTLIST class persistence-capable-superclass CDATA #IMPLIED>

<!ELEMENT field ((collection|map|array)?, (extension)*)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field persistence-modifier (persistent|transactional|none) #IMPLIED>
<!ATTLIST field primary-key (true|false) "false">
<!ATTLIST field null-value (exception|default|none) "none">
<!ATTLIST field default-fetch-group (true|false) #IMPLIED>
<!ATTLIST field embedded (true|false) #IMPLIED>

<!ELEMENT collection (extension)*>
<!ATTLIST collection element-type CDATA #IMPLIED>
<!ATTLIST collection embedded-element (true|false) #IMPLIED>

<!ELEMENT map (extension)*>
<!ATTLIST map key-type CDATA #IMPLIED>
<!ATTLIST map embedded-key (true|false) #IMPLIED>
<!ATTLIST map value-type CDATA #IMPLIED>
```

```
<!ATTLIST map embedded-value (true|false) #IMPLIED>

<!ELEMENT array (extension)*>
<!ATTLIST array embedded-element (true|false) #IMPLIED>

<!ELEMENT extension (extension)*>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>
```

Abbildung 29„DTD-File jdo.dtd von der JDO-Spezifikation“

6.15Quellenverzeichnis

[Baum94] Baumeister, M.: Ein Transaktionskonzept für die deduktive Objektbank ConceptBase, Diplomarbeit, Fachbereich Informatik, RWTH Aachen, 1994
http://www-i5.informatik.rwth-aachen.de/mbp/dipl/dipl.html

[Balz99] Balzert, H.: Lehrbuch der Objektmodellierung. Spektrum Akademischer Verlag, Heidelberg, 1999

[Russ02] Russel, C.: Java Data Objects JSR12 Version 1.0, JDO-Spezifikation, Sun Microsystems Inc., März 2002
http://jcp.org/en/jsr/detail?id=012

[Roos02] Roos, R.M.: Java Data Objects, Addison-Wesley, London, Oktober 2002
http://www.ogilviepartners.com

[Comm02] Online-Community über JDO
http://www.JDOcentral.com

[Petk01] Petkovic, D. und Brüderl, M.: Java in Datenbanksystemen, Addison-Wesley, München, November 2001

[Saak00] Sattler, K.U. und Saake, G.: Datenbanken und Java, Dpunkt Verlag, Heidelberg, 2000

[Oech01] Oechsle, R.: Verteilte Systeme, Vorlesungsskript, Fachbereich Angewandte Informatik, FH Trier, 2001

[MySQ03] MySQL: MySQL-Datenbank
http://www.mysql.de/downloads/mysql-3.23.html

[Conn03] JDBC-Treiber: Connector/J V3.0
http://www.mysql.de/downloads/api-jdbc-stable.html

[MaMa02] JDBC-Treiber: Mark Matthews JDBC-Treiber
http://www.mysql.de/downloads/api-jdbc.html

[Sign02] Signsoft: Benutzerhandbuch intelliBO 3.1, Signsoft GmbH , Dresden, 2002
http://www.signsoft.com/downloads/intellibo/3.1/documents/userguide_de.pdf

[Borl03] Borland: Profiler OptimizeIt, Borland Software Corporation, Kalifornien
http://www.borland.com/optimizeit

[Hibe03] Sourgeforge: Open-Source Objekt-Relational-Mapper Hibernate, Open Source Development Network
http://hibernate.bluemars.net

[Cast03] Castor: Open-Source Databinding-Framework Castor, Exolab Group,
http://castor.exolab.org/

[Hype03] Hypersonic SQL: Open-Source Datenbank Hypersonic SQL, Hypersonic SQL Group
http://hsqldb.sourceforge.net/

[Kodo03] SolarMetric: Benutzerhandbuch Kodo JDO, SolarMetric, Washington D.C., 2003
http://www.solarmetric.com/

[Lido03] Libelis: Benutzerhandbuch Lido, Libelis, Paris, 2003
http://www.libelis.com/